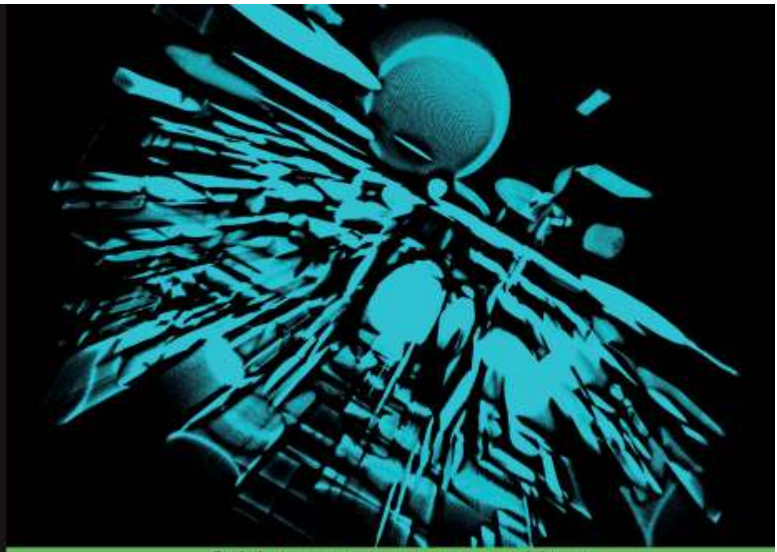


Quick answers to common problems

PlayStation®Mobile Development Cookbook

Over 65 recipes that will help you create and develop amazing mobile applications!

PlayStation.Mobile

Michael Fleischauer

[PACKT] PUBLISHING

Table of Contents

PlayStation®Mobile Development Cookbook
Credits
About the Author
About the Reviewers
www.PacktPub.com
    Support files, eBooks, discount offers and more
        Why Subscribe?
        Free Access for Packt account holders
Preface
    A tour of the PlayStation Mobile SDK
        PSM Studio IDE
        Compiler and runtime
        UIComposer
        Other utilities
        PlayStation Mobile certified devices
            PlayStation Vita specifications
            HTC Hero One X specifications
    What this book covers
    What you need for this book
    Who this book is for
    Conventions
    Reader feedback
    Customer support
        Downloading the example code
        Downloading the color images of this book
        Errata
        Piracy
        Questions
1. Getting Started
    Introduction
    Accessing the PlayStation Mobile portal
        Getting ready
        How to do it...
        How it works...
    Installing the PlayStation Mobile SDK

- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Creating a simple game loop
- Getting ready
- How to do it...
- How it works...
- There's more...

Loading, displaying, and translating a textured image
- Getting ready
- How to do it...
- How it works...
- There's more...

"Hello World" drawing text on an image
- Getting ready
- How to do it...
- How it works...
- There's more...

Deploying to PlayStation Mobile certified Android devices
- Getting ready
- How to do it...
- There's more...

Deploying to a PlayStation Vita
- Getting ready
- How to do it...
- There's more...
- See also

Manipulating an image dynamically
- Getting ready
- How to do it...
- How it works...
- There's more...

Working with the filesystem
- Getting ready
- How to do it...
- How it works...
- There's more...
- See also

Handling system events
Getting ready
How to do it...
How it works...
2. Controlling Your PlayStation Mobile Device
Introduction
Handling the controller's d-pad and buttons
Getting ready
How to do it...
How it works...
There's more...
See also
Using the Input2 wrapper class
Getting ready
How to do it...
How it works...
There's more...
See also
Using the analog joysticks
Getting ready
How to do it...
How it works...
There's more...
See also
Handling touch events
Getting ready
How to do it...
How it works...
There's more...
See also
Using the motion sensors
Getting ready
How to do it...
How it works...
There's more...
See also
Creating onscreen controls for devices without gamepads
Getting ready
How to do it…
How it works...

There's more...
See also
Configuring an Android application to use onscreen controls
Getting ready
How to do it…
How it works...
There's more...
See also
3. Graphics with GameEngine2D
Introduction
A game loop, GameEngine2D style
Getting ready
How to do it...
How it works...
There's more...
See also
Creating scenes
Getting ready
How to do it...
How it works...
There's more...
Adding a grid
See also
Adding a sprite to a scene
Getting ready
How to do it...
How it works...
There's more...
See also
Creating a sprite sheet
Getting ready
How to do it...
How it works...
There's more...
See also
Using a sprite sheet in code
Getting ready
How to do it...
How it works...
There's more...

See also
Batching a sprite with SpriteLists
Getting ready
How to do it...
How it works...
There's more...
See also
Manipulating a texture's pixels
Getting ready
How to do it...
How it works...
There's more...
See also
Creating a 2D particle system
Getting ready
How to do it...
How it works...
There's more...
See also
4. Performing Actions with GameEngine2D
Introduction
Handling updates with Scheduler
Getting ready
How to do it...
How it works...
There's more...
See also
Working with the ActionManager object
Getting ready
How to do it...
How it works...
There's more...
See also
Using predefined actions
Getting ready
How to do it...
How it works...
There's more...
See also
Transitioning between scenes

Getting ready
How to do it...
How it works...
There's more...
See also
Simple collision detection
Getting ready
How to do it...
How it works...
There's more...
See also
Playing background music
Getting ready
How to do it...
How it works...
There's more...
See also
Playing sound effects
Getting ready
How to do it...
How it works...
There's more...
See also
5. Working with Physics2D
Introduction
Creating a simple simulation with gravity
Getting ready
How to do it...
How it works...
There's more...
See also
Switching between dynamic and kinematic
Getting ready
How to do it...
How it works...
There's more...
See also
Creating a (physics!) joint
Getting ready
How to do it...

How it works...
There's more...
See also
Applying force and picking a physics scene object
Getting ready
How to do it...
How it works...
There's more...
See also
Querying if a collision occurred
Getting ready
How to do it...
How it works...
There's more...
See also
Rigid body collision shapes
Getting ready
How to do it...
How it works...
There's more...
Building and using an external library
Getting ready
How to do it...
How it works...
There's more...
6. Working with GUIs
Introduction
"Hello World" – HighLevel.UI style
Getting ready
How to do it...
How it works...
There's more...
See also
Using the UI library within a GameEngine2D application
Getting ready
How to do it...
How it works...
There's more...
See also
Creating and using hierarchies of widgets

Getting ready
How to do it...
How it works...
There's more...
Container widgets
Common widgets
Creating a UI visually using UIComposer
Getting ready
How to do it...
How it works...
There's more...
See also
Displaying a MessageBox dialog
Getting ready
How to do it...
How it works...
There's more...
See also
Handling touch gestures and using UI effects
Getting ready
How to do it...
How it works...
There's more...
See also
Handling language localization
Getting ready
How to do it...
There's more...
See also
7. Into the Third Dimension
Introduction
Creating a simple 3D scene
Getting ready
How to do it...
How it works...
There's more...
See also
Displaying a textured 3D object
Getting ready
How to do it...

How it works...
There's more...
See also
Implementing a simple camera system
Getting ready
How to do it...
How it works...
See also
A fragment (pixel) shader in action
Getting ready
How to do it...
How it works...
There's more...
See also
A vertex shader in action
Getting ready
How to do it...
How it works...
There's more...
See also
Adding lighting to your scene
Getting ready
How to do it...
How it works...
There's more...
See also
Using an offscreen frame buffer to take a screenshot
Getting ready
How to do it...
How it works...
There's more...
See Also
8. Working with the Model Library
Introduction
Importing a 3D model for use in PlayStation Mobile
Getting ready
How to do it...
How it works...
There's more...
See also

Loading and displaying a 3D model
Getting ready
How to do it...
How it works...
There's more...
See also
Using BasicProgram to perform texture and shader effects
Getting ready
How to do it...
How it works...
There's more...
See also
Controlling lighting using BasicProgram
Getting ready
How to do it...
How it works...
There's more...
See also
Animating a model
Getting ready
How to do it...
How it works...
There's more...
See also
Handling multiple animations
Getting ready
How to do it...
How it works...
There's more...
See also
Using bones to add a sword to our animated model
Getting ready
How to do it...
How it works...
There's more...
9. Finishing Touches
Introduction
Opening and loading a web browser
Getting ready
How to do it...

How it works...
There's more...
Socket-based client and server networking
Getting ready
How to do it...
How it works...
There's more...
See Also
Accessing (Twitter) data over the network using REST and HttpWebRequest
Getting ready
How to do it...
How it works...
There's more...
See also
Copying and pasting using Clipboard
Getting ready
How to do it...
How it works...
There's more...
Embedding and retrieving a resource from the application assembly
Getting ready
How to do it...
How it works...
There's more...
See also
Configuring your application using PublishingUtility
Getting ready
How to do it...
There's more…
See also
Creating downloadable content (DLC) for your application
Getting ready
How to do it...
How it works...
There's more...
See also
A. Publishing Your Application
Introduction

Process overview
Obtaining a developer license
Your publisher keys
Preparing your application for publication
Publishing your application
Index

PlayStation®Mobile Development Cookbook

PlayStation®Mobile Development Cookbook

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

"PlayStation" is a registered trademark of Sony Computer Entertainment Inc.

" " is a trademark of the same company.

First published: March 2013

Production Reference: 1180313

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-84969-418-6

www.packtpub.com

Cover Image by Suresh Mogre
(<suresh.mogre.99@gmail.com>)

Credits

Author

Michael Fleischauer

Reviewers

Neil Brown

Mehul Shukla

Acquisition Editor

Erol Staveley

Lead Technical Editor

Erol Staveley

Technical Editors

Sharvari Baet

Devdutt Kulkarni

Kirti Pujari

Project Coordinator

Anurag Banerjee

Proofreader

Lawrence A. Herman

Indexer

Rekha Nair

Graphics

Aditi Gajjar

Production Coordinator

Manu Joseph

Cover Work

Manu Joseph

About the Author

Michael Fleischauer has spent the last 16 years working as a programmer in a number of different industries from 3D tools creation to automotive and banking. Most recently he launched the internet start-up Flexamail. In his spare time he writes for and runs the game development site GameFromScratch.com, a popular destination for game development tutorials and news. Michael was recently made the first PlayStation Mobile MVP by Sony. Michael lives in Toronto, Canada with his wife and daughter.

I would like to thank my daughter Kailyn for sending me down this new career path and my wife Jenn for supporting me through it all. My thanks to my editor Erol Stavely and the entire team at Packt Publishing; this entire experience has been a pleasant one. Finally, I would like to thank Paul Holman, Mehul Shukla, and the entire PlayStation Mobile team at Sony; your ongoing support is greatly appreciated!

About the Reviewers

Neil Brown is Senior Team Leader in the SCEE R & D Developer Services team. Apart from providing technical support and performance advice, he coordinates support for all PlayStation platforms in the historic PAL regions, including PlayStation Mobile.

Neil has given technical talks at a number of games industry conferences around the world for SCE, speaking about PSM at Develop Brighton, Casual Connect in Kiev, and Nordic Game.

Neil has been in the games industry for almost 10 years, and has Masters degrees in Software Engineering, and Physics with Astrophysics.

Mehul Shukla is one of the PlayStation®Mobile specialists in the SCEE R & D Developer Services team. The Developer Services team provides front-line engineering support for all game developers, large or small, on all PlayStation platforms. On a daily basis, he provides technical support and performance advice for developers all over the globe on the PSM community forums.

Mehul has also given technical talks about PSM at a number of games industry conferences and academic events.

Mehul joined SCEE R & D straight from University and has a Master's degree in Games programming and a Bachelor's degree in Computer Systems Engineering.

I would like to thank Mike for his involvement in PlayStation®Mobile and his contribution to the developer community. Mike is one of the most valuable members of the PlayStation®Mobile community and has been actively involved in providing useful advice on our developer forums. We wish him all the best in the future.

Packt Publishing would also like to thank Paul Holman, Marijke

Coopmans, and Sarah Thomson for their help and support throughout the development of this book.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content

- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Preface

The PlayStation Mobile SDK presents an incredible opportunity for developers to easily and affordably create and sell applications for the PlayStation Vita, as well as a number of PlayStation certified devices. This represents the first time it has been possible to write applications for a console quality portable device without first having to spend several thousands of dollars on professional development kits.

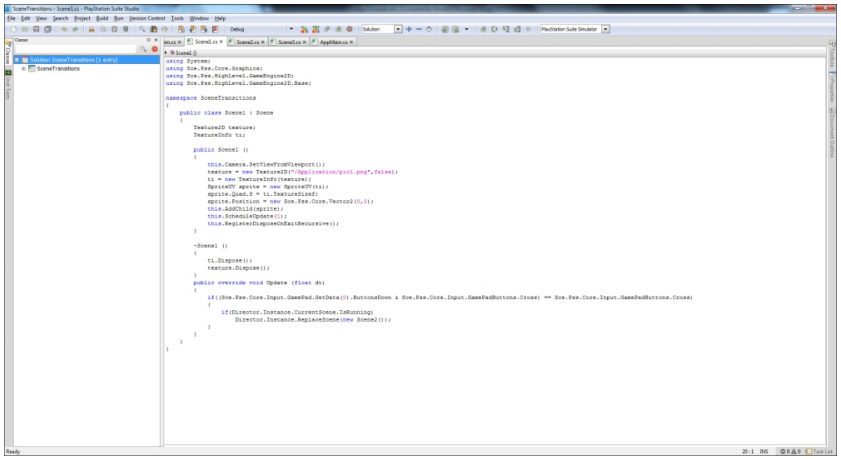
It includes all of the tools you require to successfully create a game, including a complete **Integrated Development Environment** (**IDE**), a C#/Mono based compiler and runtime, as well as the tools and utilities required to create interfaces and import game assets. The SDK is suitable for a range of developers, from hobbyists to Indie game developers as well as AAA game studios. A number of large studios, including From Software, Gameloft, and Sega, have announced their support for PlayStation Mobile. To date, a number of titles have already shipped and are available in the online store.

A tour of the PlayStation Mobile SDK

We will now take a quick tour of what is included in the SDK; if you haven't already, download it from the PlayStation Mobile Developer Portal at https://psm.playstation.net/. The SDK includes the components that we will discuss now.

PSM Studio IDE

The PSM Studio is a complete IDE derived from the popular open source MonoDevelop project. It includes a complete code editor, project management system, and integrated debugger. It contains most features you would expect of a modern IDE such as unit testing, code completion, and refactoring.



Compiler and runtime

PlayStation Mobile is built on top of the Mono compiler and virtual machine. In addition to the PlayStation provided libraries, it includes the following `.NET` libraries:

- `System`
- `System.Core`
- `System.Runtime.Serialization`
- `System.ServiceModel`
- `System.ServiceModel.Web`
- `System.Web.Services`
- `System.Xml`
- `System.Xml.Linq`

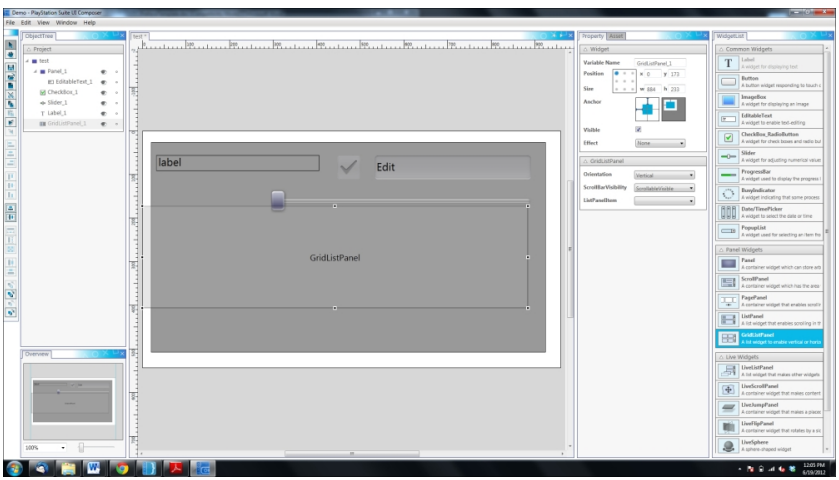
In addition to those standard `.NET` libraries, Sony has provided the following libraries:

- `Sce.Pss.Core`
- `Sce.Pss.HighLevel.GameEngine2D`
- `Sce.Pss.HighLevel.Model`
- `Sce.Pss.HighLevel.Physic2D`
- `Sce.Pss.HighLevel.UI`

You can also make use of any existing C# code that does not require native access. We will look at each of these libraries in more detail throughout the book.

UIComposer

The UIComposer enables you to visually create user interfaces. It includes a comprehensive set of widgets including buttons, text fields, progress bars, flip panels, scrolling areas, and more. Ultimately UIComposer is a code generator that will output a `.cs` file that makes use of partial classes to keep your application logic separate from system generated code. If you are familiar with WinForms, this will be instantly comfortable for you. It is a drag-and-drop environment, enabling you to build your user interfaces in a visual manner:



Other utilities

The SDK includes a number of utilities for importing your various assets for use in your game. There is a command line based model converter for importing your 3D model into PSM's native MDX format. There are also tools for importing Flash animations, and graphical shaders, as well as a tool for creating on-screen controllers for Android devices. Finally, there is the PublishingUtility, which is used to prepare your application for deployment to the online store as well as for creating downloadable content. Assuming a default installation, all these tools and more are located in the folder `C:\Program Files(x86)\SCE\PSM\tools`. We will cover many of these tools in detail later in the book.

PlayStation Mobile certified devices

PlayStation Mobile can target the PlayStation Vita, as well as a growing number of PlayStation certified devices. Currently this includes a number of Xperia mobile phones, Sony Android tablets, and a series of HTC phones. You can see a full list of certified phones at http://www.playstation.com/psm/certified.html.

It is hard to believe the level of technology being packed into these devices. Let us now see the specifications for the PlayStation Vita and HTC One X phones, two supported devices.

PlayStation Vita specifications

The following are the system requirements for PlayStation Vita:

- ARM A9 Quad Core processor
- PowerVR SGX543MP4 Quad Core GPU
- 512 MB RAM and 128 MB Video RAM
- 5" 960x544 pixel multi-touch display
- GPS, two cameras, two touch sensors, gyroscope, dual analog sticks

HTC Hero One X specifications

The following are the system requirements for HTC Hero One X:

- ARM A9 Dual or Quad Core Processor (depending on region)
- NVidia Tegra3
- 1024 MB RAM with 16-32 GB of storage
- 4.7" 1280 x 720 pixel multi-touch display
- GPS, Gyroscope, G-Sensor, Proximity Sensor, two cameras

As you can see, PlayStation Mobile is running on some remarkably capable hardware. It’s hard to believe how far things have come when you consider the original PSP was running on a single CPU running at 333 MHz with only 32 MB RAM while the Gameboy DS was powered by a pair of CPUs running at 67 and 33.5 MHz,

respectively, with a paltry 4 MB of RAM. This generation of handheld devices is sporting hardware comparable to what is found in the PlayStation 3 and Xbox 360!

What this book covers

Chapter 1, *Getting Started*, covers getting the PlayStation Mobile SDK up and running and deployed to various devices (a task of some complexity). It jumps right in, creating basic graphical applications and covers the details and restraints of working on the devices.

Chapter 2, *Controlling Your PlayStation Mobile Device*, covers all the various ways in which you can control PSM devices, from the traditional joystick to touch and motion controls. Additionally, since not all devices have the same capabilities, it covers creation and handling of on-screen controllers.

Chapter 3, *Graphics with GameEngine 2D*, covers the graphical aspects of working with GameEngine2D—a higher level 2D gaming engine similar in design to the popular Cocos2D. It covers all aspects of 2D graphics from scenes and sprites to special effects and performance optimizations with SpriteLists.

Chapter 4, *Performing Actions with GameEngine 2D*, covers the action side of using GameEngine2D. This involves updating game objects, scheduling events, and executing actions, both in-built actions such as MoveTo and MoveBy and also defining your own.

Chapter 5, *Working with Physics2D*, covers working with Physics2D, PSM SDK's in-built 2D physics system for creating physics simulations. Physics2D is not the only option for physics, so it also looks at integrating the popular BEPU and FarSeer XNA physics engines into your PSM application.

Chapter 6, *Working with GUIs*, covers the UI system built into the PlayStation Mobile. This ranges from creating on-screen buttons and panels, handling clicks and hold events, to advanced touch gestures. Additionally, it covers using UIComposer to visually create and edit UIs.

Chapter 7, *Into the Third Dimension*, covers working in 3D, from creating a camera and using graphic primitives to using fragment

and vertex shaders.

Chapter 8, *Working with the Model Library*, covers working with 3D objects, including creating and exporting them using a third party application, converting them using the SDK tools, and finally displaying and animating them in 3D.

Chapter 9, *Finishing Touches* covers the wealth of networking options available to PSM devices. Additionally, we cover the Publishing tool and preparing your application for deployment to the PlayStation Mobile App Store.

Appendix, *Publishing Your Application*, covers the process of compiling, signing, packaging, and deploying your finished application to the PlayStation App Store.

What you need for this book

In order to get the most out of this book, you need to have a Windows computer capable of running the PlayStation Mobile SDK. You will also need a copy of the PlayStation Suite SDK, which can be downloaded at http://psm.playstation.net/.

Most samples can be run using the included simulator, but to get the most out of the PlayStation Mobile SDK, you should have a hardware device to run on, such as a PlayStation Vita or a PlayStation certified Android device. Currently, it is free to use the simulator, but not to deploy to a device.

The PlayStation Mobile Studio has the following system requirements:

- One of the following operating systems:
    - Microsoft® Windows® XP Service Pack 3 or later (32 bit version only)
    - Microsoft® Windows® 7 Service Pack 1 (32 bit or 64 bit version) or later
- 3 GHz processor or greater
- At least 2 GB of free space on your hard disk
- At least 4 GB of RAM
- A graphics card that supports OpenGL 3.0 or higher
- A sound card compatible with DirectX 9.0
- 1 or more USB 2.0 compatible ports

Who this book is for

If you've got some prior experience with C# and want to create awesome projects for the PlayStation®Vita and PlayStation™ Certified devices then this book is for you.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

A block of code is set as follows:

```
Director.Initialize();
Scene scene = new Scene();
scene.Camera.SetViewFromViewport();
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

> **Note**
>
> Warnings or important notes appear in a box like this.

> **Tip**
>
> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to <feedback@packtpub.com>, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from http://www.packtpub.com/sites/default/files/downloads/4186OT_ColoredImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/support, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Getting Started

In this chapter we will cover:

- Accessing the PlayStation Mobile portal
- Installing the PlayStation Mobile SDK
- Creating a simple game loop
- Loading, displaying, and translating a textured image
- "Hello World" drawing text on an image
- Deploying to PlayStation Mobile certified Android devices
- Deploying to a PlayStation Vita
- Manipulating an image dynamically
- Working with the filesystem
- Handling system events

Introduction

The PlayStation Mobile (PSM) SDK represents an exciting opportunity for game developers of all stripes, from hobbyists to indie and professional developers. It contains everything you need to quickly develop a game using the C# programming language. Perhaps more importantly, it provides a market for those games. If you are currently using XNA, you will feel right at home with the PSM SDK.

You may be wondering at this point, *Why develop for PlayStation Mobile at all*? Obviously, the easiest answer is, *so you can develop for PlayStation Vita*, which of itself will be enough for many people. Perhaps, though the most important reason is that it represents a group of dedicated gamers hungry for games. While there are a wealth of games available for Android, finding them on the App Store is a mess, while supporting the literally thousands of devices is a nightmare. With PlayStation Mobile, you have a common development environment, targeting powerful devices with a dedicated store catering to gamers.

We are now going to jump right in and get those tools up and running. Of course, we will also write some code and show how

easy it is to get it running on your device. PlayStation Mobile allows you to target a number of different devices and we will cover the three major targets (the Simulator, PlayStation Vita, and Android). You do not need to have a device to follow along, although certain functionality will not be available on the Simulator.

One thing to keep in mind with the PlayStation Mobile SDK is that it is essentially two SDKs in one. There is a much lower level set of libraries for accessing graphics, audio, and input, as well as a higher-level layer build over the top of this layer, mostly with the complete source available. Of course, underneath this all there is the .NET framework. In this chapter, we are going to deal with the lower level graphics interface. If the code seems initially quite long or daunting for what seems like a simple task, don't worry! There is a much easier way that we will cover later in the book.

Accessing the PlayStation Mobile portal

This recipe looks at creating a PSM portal account. For this process it is mandatory to download and use the PSM SDK.

Getting ready

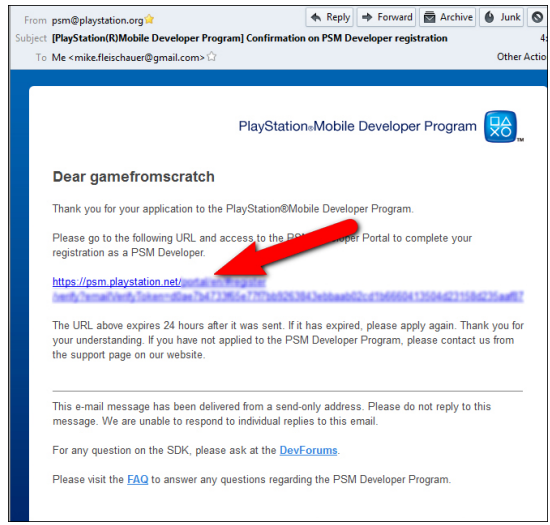
You need to have a Sony Entertainment Network (SEN) account to register with the PSM portal. This is the standard account you use to bring your PlayStation device online, so you may already have one. If not, create one at http://bit.ly/Yiglfk before continuing.

How to do it...

1. Open a web browser and log in to http://psm.playstation.net. Locate and click on the **Register** button.

2. Sign in using the SEN account.
3. Agree to the Terms and Conditions. You need to scroll to the bottom of the text before the **Agree** button is enabled. But, you always read the fine print anyways... don't you?
4. Finally select the e-mail address and language you want for the PlayStation Mobile portal. You can use the same e-mail you used for your SEN account. Click on **Register**.
5. An e-mail will be sent to the e-mail account you used to

sign up. Locate the activation link and either click on it, or copy and paste into a browser window:



6. Your account is now completed, and you can log in to the PSM developer portal now.

How it works...

A PlayStation Mobile account is mandatory to download the PSM tools. Many of the links to the portal require you to be logged in before they will work. It is very important that you create and activate your account and log in to the portal before continuing on with the book! All future recipes assume you are logged in to the portal.

Installing the PlayStation Mobile SDK

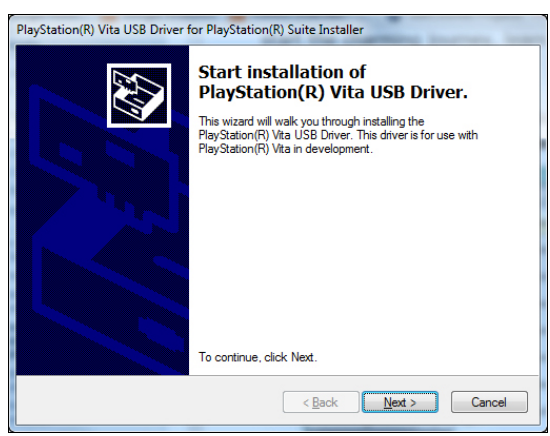
This recipe demonstrates how to install the PlayStation Mobile SDK.

Getting ready

First you need to download the PlayStation Mobile SDK; you can download it from http://bit.ly/W8rhhx.

How to do it...

1. Locate the installation file you downloaded earlier and double-click to launch the installer. Say yes to any security related questions.
2. Take the default settings when prompting, making sure to install the runtimes and `GTK#` libraries.
3. The installer for the Vita drivers will now launch. There is no harm in installing them even if you do not have a Vita:



4. Installation is now complete; a browser window with the current release notes will open.

How it works...

The SDK is now installed on your machines. Assuming you used default directories, the SDK will be installed to `C:\Program Files (x86)\SCE\PSM` if you are running 64 bit Windows, or to `C:\Program Files\SCE\PSM` if you are running 32 bit Windows. Additionally all of the documentation and samples have been installed under the Public account, located in `C:\Users\Public\Documents\PSM`.

There's more...

There are a number of samples available in the `samples` directory and you should certainly take a moment to check them out. They range in complexity from simple Hello World applications, up to a full blown 3rd person 3D role playing game (RPG). They are, however, often documented in Japanese and often rely on other samples, making learning from them a frustrating experience at times, at least, for those of us who do not understand Japanese!

See also

- See the *A Tour of the PlayStation Mobile SDK* section in the *Preface* for a better understanding of what is included in the SDK you just installed

Creating a simple game loop

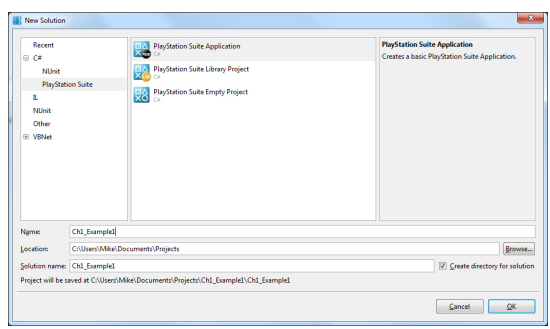
We are now going to create our first PSM SDK application, which is the main loop of your application. Actually all the code in this sample is going to be generated by PSM Studio for us.

Getting ready

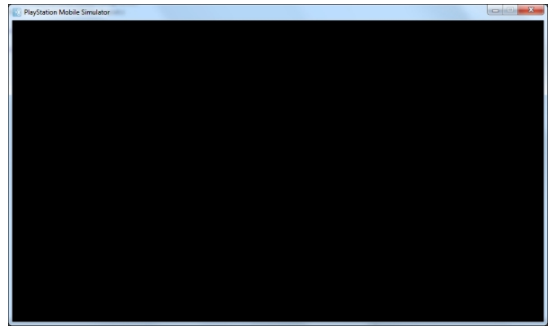
From the start menu, locate and launch PSM Studio in the `PlayStation Mobile` folder.

How to do it...

1. In PSM Studio, select the **File** | **New** | **Solution...** menu.
2. In the resulting dialog box, in the left-hand panel expand **C#** and select **PlayStation Suite**, then in the right-hand panel, select **PlayStation Suite Application**. Fill in the **Name** field, which will automatically populate the **Solution name** field. Click on **OK**.



3. Your workspace and boilerplate code will now be created; hit the *F5* key or select the **Run** | **Start Debugging** menu to run your code in the Simulator.



Not much to look at, but it's your first running PlayStation Mobile application! Now let's take a quick look at the code it generated:

```
using System;
using System.Collections.Generic;
using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Environment;
using Sce.PlayStation.Core.Graphics;
using Sce.PlayStation.Core.Input;

namespace Ch1_Example1
{
  public class AppMain{
    private static GraphicsContext
graphics;

    public static void Main (string[]
args){
      Initialize ();

      while (true) {
        SystemEvents.CheckEvents ();
        Update ();
        Render ();
      }
    }

    public static void Initialize (){
```

```
            graphics = new GraphicsContext
();
        }

        public static void Update (){
            var gamePadData =
GamePad.GetData (0);
        }

        public static void Render ()
{
            graphics.SetClearColor (0.0f,
0.0f, 0.0f, 0.0f);
            graphics.Clear ();
            graphics.SwapBuffers ();
        }
    }
}
```

> **Tip**
>
> **Downloading the example code**
>
> You can download the example code files for all Packt Publishing books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

How it works...

This recipe shows us the very basic skeleton of an application. Essentially it loops forever, displaying a black screen.

```
        private static GraphicsContext
        graphics;
```

The `GraphicsContext` variable represents the underlying OpenGL context. It is used to perform almost every graphically related action. Additionally, it contains the capabilities (resolution, pixel depth, and so on) of the underlying graphics device.

All C# based applications have a `main` function, and this one is no exception. Within `Main()` we call our `Initialize()` method, then loop forever, checking for events, updating, and finally rendering the frame. The `Initialize()` method simply creates a new `GraphicsContext` variable. The `Update()` method polls the first gamepad for updates (we will cover controls in more detail later).

Finally `Render()` uses our `GraphicsContext` variable to first clear the screen to black using an RGBA color value, then clears the screen and swaps the buffers, making it visible. Graphic operations in PSM SDK generally are drawn to a back buffer.

There's more...

The same process is used to create PlayStation Suite library projects, which will generate a DLL file. You can use almost any C# library that doesn't rely on native code (`pInvoke` or `Unsafe`); however, they need to be recompiled into a PSM compatible DLL format.

Color in the PSM SDK is normally represented as an RGBA value. The RGBA acronym stands for red, green, blue, and alpha. Each is an `int` variable type, with values ranging from 0 to 255 representing the strength of each primary color. Alpha represents the level of transparency, with 0 being completely transparent and 256 being opaque.

Loading, displaying, and translating a textured image

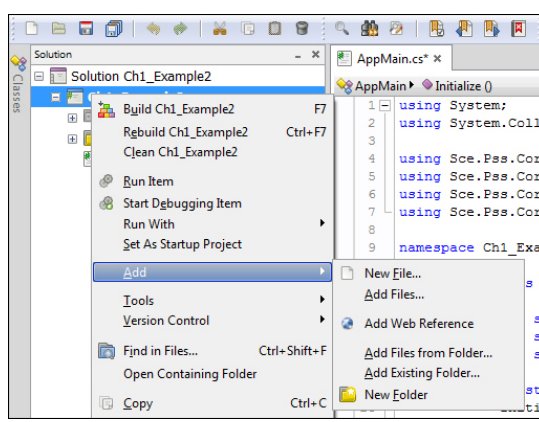
This recipe is going to create an application that loads a texture from an image file and displays it centered on the screen. This example is actually rather daunting, throwing quite a bit of information at a new developer. Don't worry if it seems overly complex for now; by the end of the book it will make more sense. If you feel overwhelmed, I recommend you continue the book and revisit this recipe later.

Getting ready

Following the instructions presented in the *Creating a simple game loop* recipe, create a new solution. I have named mine as `Ch1_Example2`.

How to do it...

1. First, we need to add an image file to our project to use as a texture. This can be done by right-clicking our project in the **Solution** panel and selecting **Add** | **Add Files...** from the menu, as shown in the following screenshot:



2. Now, we need to tell PSM Studio what to do with this file. Select our newly added image, right-click on it, and select **Build Action** | **Content**.
3. Now, enter the following code:

```
using System;
using
System.Collections.Generic;

using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Environment;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.Core.Input;

namespace Ch1_Example2
{
  public class AppMain
  {
    private static
GraphicsContext _graphics;
    private static Texture2D
_texture;
    private static
VertexBuffer _vertexBuffer;
    private static
ShaderProgram
_textureShaderProgram;
    private static Matrix4
_localMatrix;
    private static Matrix4
_projectionMatrix;
    private static Matrix4
_viewMatrix;
    private static float
_viewportWidth;
    private static float
_viewportHeight;
```

```
    public static void Main
(string[] args){
      Initialize ();

      while (true) {

SystemEvents.CheckEvents ();
        Update ();
        Render ();
      }
    }

    public static void
Initialize (){
      _graphics = new
GraphicsContext ();
      _viewportWidth =
_graphics.GetFrameBuffer().Width;
      _viewportHeight =
_graphics.GetFrameBuffer().Height;

      _texture = new
Texture2D("/Application/
FA-18H.png",false);
      _vertexBuffer = new
VertexBuffer(4,VertexFormat.Float3,Vertex

_vertexBuffer.SetVertices(0,new
float[] {
        1,0,0,
        _texture.Width,0,0,

_texture.Width,_texture.Height,0,

0,_texture.Height,0});


_vertexBuffer.SetVertices(1,new
float[]{
```

```
            0.0f,0.0f,
            1.0f,0.0f,
            1.0f,1.0f,
            0.0f,1.0f});

        _textureShaderProgram
= new ShaderProgram("/
Application/shaders/
Texture.cgx");

        _projectionMatrix =
Matrix4.Ortho(
            0,_viewportWidth,
            0,_viewportHeight,
            0.0f,32768.0f);

        _viewMatrix =
Matrix4.LookAt(
            new
Vector3(0,_viewportHeight,0),
            new
Vector3(0,_viewportHeight,1),
            new
Vector3(0,-1,0));

        _localMatrix =
Matrix4.Translation(new
Vector3(-_texture.Width/2,-
_texture.Height/2,0.0f))
        *
Matrix4.Translation(new
Vector3(_viewportWidth/2,_viewportHeight/
}

    public static void
Update (){
        var gamePadData =
GamePad.GetData (0);
    }
```

```
    public static void
Render (){

_graphics.SetClearColor
(0.0f, 0.0f, 0.0f, 0.0f);
      _graphics.Clear ();

      var
worldViewProjection =
_projectionMatrix *
_viewMatrix * _localMatrix;

_textureShaderProgram.SetUniformValue(0,r
worldViewProjection);


_graphics.SetShaderProgram(_textureShader

_graphics.SetVertexBuffer(0,_vertexBuffer

_graphics.SetTexture(0,_texture);


_graphics.DrawArrays(DrawMode.TriangleFar

      _graphics.SwapBuffers
();
    }
  }
}
```

How it works...

Phew! That sure seemed like a lot of code to simply display a single image on screen, didn't it? The truth is, you did a lot more than just load and draw a texture. Let's jump in and look at exactly what we just created.

First, we declared the following variables, in addition to our existing

`GraphicsContext` variable:

- `_texture` is our `Texture2D` object that is going to hold our textured image.
- `_vertexBuffer` is a `VertexBuffer` object that holds the 3D quad geometry we are going to map our texture on.
- `_shaderProgram` is a `ShaderProgram` variable, the texture shader needed to render our texture. The `GraphicsContext` variable requires at least one. Fortunately, a simple one with the extension `.cgx` was created for you already by PSM Studio when you created the project.
- `_localMatrix`, `_projectionMatrix`, and `_viewMatrix` are `Matrix4` objects, representing the textured object's position.
- `_viewportWidth` and `_viewportHeight` contain the dimensions of our window.

The bulk of our activity is in the `Initialize()` method. Once again, we create a `GraphicsContext` variable, and then store the dimensions of the frame buffer in the `_viewportHeight` and `_viewportWidth` variables. Next, we create our `Texture2D` object, passing the constructor the filename and whether or not we want a mipmap generated.

Next, we create a `_vertexBuffer` object, which is going to be a fullscreen quad we can draw our texture on. We make two calls to `SetVertices()`. The first call is defining the *x*, *y*, and *z* float variables that make up the four vertices of the fullscreen quad. The second `SetVertices` function call is four *x* and *y* texture coordinates. Texture coordinates are represented with a value from 0 to 1.

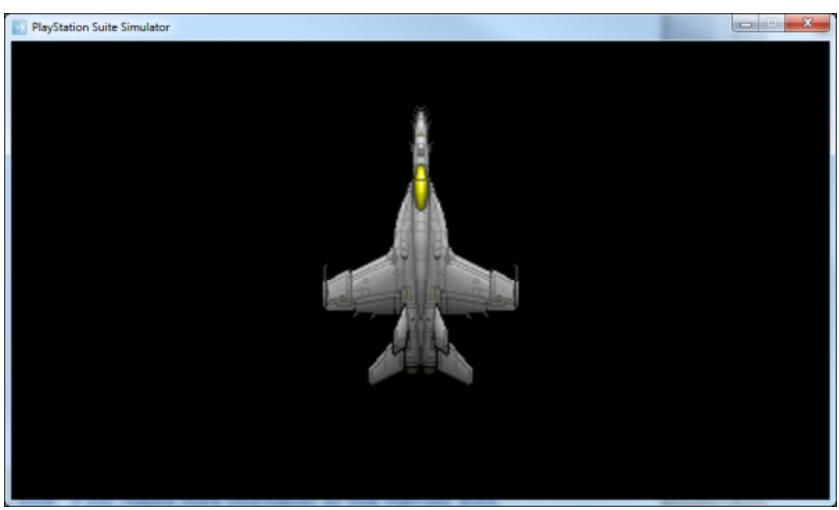
Next, we create our `_textureShaderProgram` function using the default shader PSM Studio created for us. We will cover shaders in more detail later in this chapter.

Finally, we set up the `_projectionMatrix`, `_viewMatrix`, and `_localMatrix` objects. The projection matrix is an orthographical matrix that represents our screen. The view matrix represents the

camera within the world, using `Matrix4.LookAt. LookAt()`, which requires 3 vectors, the first representing your eye's location in 3D space, the second, the 3D point you are looking at, and the third, the direction where "UP" is, in this case in the Y direction. Finally, the local matrix represents the position of texture, which we want to be centered in the middle of the screen.

Now, let's take a look at the `Render()` function, where our texture is going to be displayed to the screen. As before, we set the clear color to black and clear the buffer. Next, we generate our `worldViewProjection` matrix by multiplying our projection, view and local matrices together. We then bind our `worldViewProjection` matrix to our `shader` program and then set our `shader` program to the `GraphicsContext` variable. We also set our `VertexBuffer` object and `Texture2D` object to the `GraphicsContext` variable. The `DrawArrays()` call is what ties it all together, using our `worldViewMatrix` to transform our vertices from our `VertexBuffer` object and applying our texture map, rendering it all to the active buffer. Finally, we make that buffer visible, which draws it on screen.

Here is our program in action, rendering our sprite centered to the screen:



Again, if that seemed overly complex, don't panic! Most of this code only needs to be written once, and you have the option of not

working at this low a level if you should choose!

There's more...

Build actions will be executed when your project is compiled, copying the content to the appropriate folder, performing whatever conversions are required. If you are used to XNA, this is similar to the functionality of the content pipeline, but not programmable.

Note

Why is there 3D in my 2D?

The bulk of this example was actually going through the process of faking a 2D environment using 3D. The reason is modern GPUs are optimized to work in 3D. If you look at the code to most modern 2D libraries, they are actually working in 3D. If you were to work with native 2D graphics libraries, your performance would be abysmal.

An explanation of 3D mathematics is beyond the scope of this book, but the Kahn Academy (see http://www.khanacademy.org/) is an excellent free resource with thousands of video tutorials.

Tip

The sprite I used for this example and throughout this book is from a wonderful free sprite library made available by GameDev.net user *Prince Eugn*. You can find more information and download the sprite pack at http://bit.ly/N7CPtE.

"Hello World" drawing text on an image

This recipe dynamically creates and displays a texture with text created using the imaging APIs.

Getting ready

This recipe builds on the code from the last recipe. Add the following `using` statement to the beginning of the program code:

```
using Sce.PlayStation.Core.Imaging
```

For the complete source code for this example, see `Ch1_Example3`.

How to do it...

1. In the `Initialize()` function, instead of loading a texture from the file, we will create one using the following code:

```
Image img = new
Image(ImageMode.Rgba,new
ImageSize(500,300),new
ImageColor(0,0,0,0));
img.DrawText("Hello World",
   new
ImageColor(255,255,255,255),
   new
Font(FontAlias.System,96,FontStyle.Italic
   new
ImagePosition(0,150));
_texture = new
Texture2D(500,300,false,PixelFormat.Rgba)
```

```
_texture.SetPixels(0,img.ToBuffer());
```

2. Next, update the `Render()` method in the following code in which the bolded portions represent the changes:

```
public static void Render ()
{
 _graphics.SetClearColor
(0.0f, 0.0f, 0.0f, 0.0f);
 _graphics.Clear ();

 var worldViewProjection =
_projectionMatrix *
_viewMatrix * _localMatrix;

_textureShaderProgram.SetUniformValue(0,r
worldViewProjection);



_graphics.SetShaderProgram(_textureShader

_graphics.SetVertexBuffer(0,_vertexBuffer

_graphics.SetTexture(0,_texture);


_graphics.Enable(EnableMode.Blend);

_graphics.SetBlendFunc(BlendFuncMode.Add,
BlendFuncFactor.SrcAlpha,
BlendFuncFactor.OneMinusSrcAlpha);

_graphics.DrawArrays(DrawMode.TriangleFar

 _graphics.SwapBuffers ();
}
```

How it works...

Instead of loading a texture from an image file, we create an image dynamically. In the `Image` constructor, we pass the type of image we want created, the dimensions and the background color to fill it with.

Next, we draw on our newly created image using the `DrawText()` function, which takes as parameters the text to draw, the color to draw it in, the font to use (there is only one option, `System`) and the position to draw the text at. We then create a `Texture2D` object to hold our image. We pass its constructor the image dimensions, whether we want to generate a mipmap or not, as well as the pixel format to use. Finally, we assign the pixel data to our `_texture` object by calling `SetPixel()` function and passing in a byte array generated by calling `ToBuffer()` function on our image.

We had to make the change to the `Render()` method to support blending using the alpha channel, or background would not be properly visible through the transparent portions of the text. Run the code again without `EnableMode.Blend` enabled and your text will be illegible.

Now if we run our application, we will see the following screenshot:

There's more...

You can also load a font by name instead of using the built in system font. If you need precise control over your text positioning or size, be sure to check out the `FontMetrics` and `CharMetrics` classes in the documentation.

Deploying to PlayStation Mobile certified Android devices

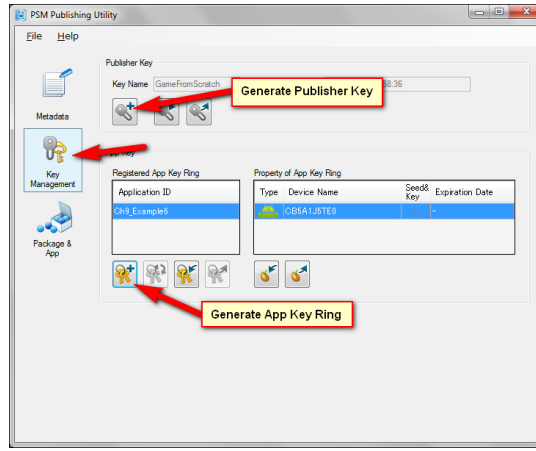
This recipe covers deploying an application to an Android device. Running on an Android device requires a developer license that you can purchase in the PSM portal.

Getting ready

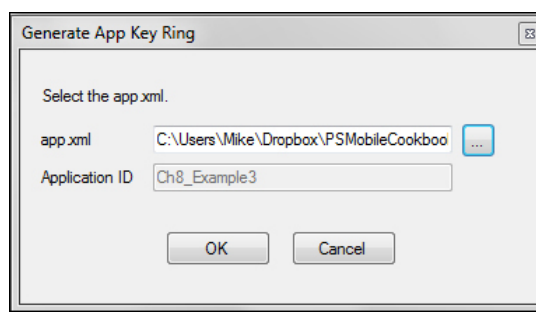
You need to have installed the PlayStation Mobile SDK to have access to required files. Of course you will also require a PlayStation Mobile compatible Android device. Make sure the Android ADB driver for your phone is installed on your computer; you can download a generic version from Google's Android development website if required.

How to do it...

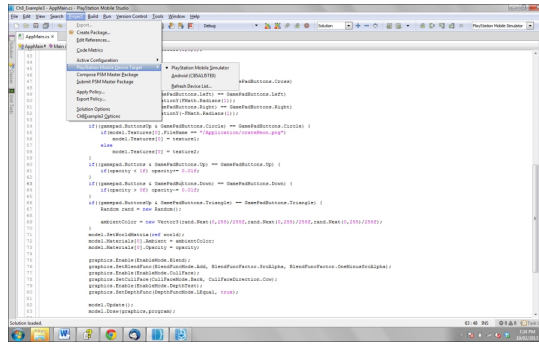
1. Attach your Android phone by USB to your computer. It may install a driver at this point, let it.
2. Open a Command Prompt (**Start** | **Run** and type `cmd.exe`) and type `"%SCE_PSM_SDK%/tools/update_psmdevassistant.bat"` including the quotes. This will install the PSM Development Assistant on your device.
3. On your device, locate and run the PSM Development Assistant application.
4. On your computer, in the `PlayStation Mobile` folder in the **Start** menu, load Publishing Utility.
5. Now you need to create a Publisher Key; you will only have to do this once. First click the **Key Management** tab, then the **Generate Publisher Key** button. Name your key and then enter your login information for the PSM portal. A key will be generated for you.



6. Click on the **Generate App Key Ring** button. In the **app.xml** field locate the `App.xml` file of the project you want to run on your device and then click on **OK**. You will have to authenticate with your SEN ID again. If all goes well, your app will now be listed in the **Registered App Key Ring** list.



7. Now fire up your project in PSM Studio. Make sure that you have the PlayStation Mobile Development Assistant running on your device and your phone is connected by a USB connection. Set Android as your target using the **Project** | **PlayStation Mobile Device Target** menu item and select your phone device.



8. Run your application by pressing the *F5* key or choosing **Run** or **Debug** from the menu and your application will be copied to your Android device and run.

There's more...

You need to make sure that USB debugging is enabled on your Android device. This varies from device to device, but is generally located in the **Settings** | **Developer Options** | **USB debugging menu**.

I ran into an error when trying to add my application to the key ring. To fix this, I recreated my Publisher Key. If you are working alone, this isn't a big process, but if you are part of a team, you will need to distribute the updated key.

Deploying to a PlayStation Vita

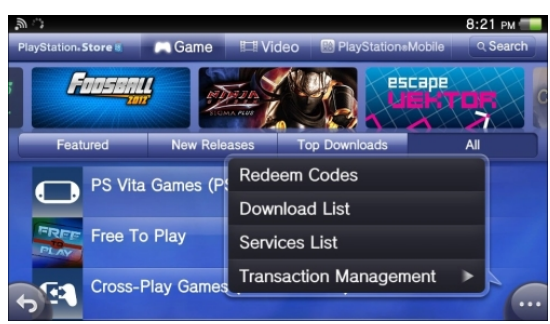
In this recipe, we will configure a PS Vita to be used with PSM Studio. Running on a PS Vita device requires a developer license, which you can purchase in the PSM portal.

Getting ready

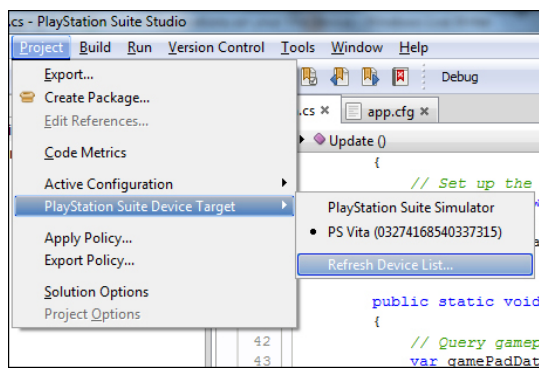
You will need to have installed the PlayStation Mobile SDK to have the required Vita drivers installed. If you did not install the drivers as part of the install process, under a default install they are available at `C:\Program Files (x86)\SCE\PSM\tools\vita\driver`. Obviously you will need to have a PS Vita and a USB cable. You will need a project to run on your Vita; load either one of the examples we created earlier or one of the examples from the SDK.

How to do it...

1. Before you can run on your Vita, you need to install the Development Assistant from the PS Store. Load the PS Store application on your PS Vita device, and then click on the **...** button in the bottom-right corner of the screen. Select **Download List**. Scroll down and select **PlayStation Mobile Development Assistant** and click on **Download**. The application will be downloaded and installed.



2. The **PS Mobile Development application** icon will now be available on your Vita. If you haven't already, connect your Vita to your PC using the USB cable. If prompted, allow it to install the driver.
3. On your Vita, run the PS Suite SDK application.
4. Make sure that your application has been added to the **Registered App Key Ring** list (see previous recipe for details).
5. Load up PSM Studio and load a solution to run on your Vita. Now open the **Project** | **PlayStation Suite Device Target** | **PS Vita** menu item. If your PS Vita doesn't show up, select **Refresh Device List...** and try again. The device will show as off, if the Development Assistant isn't running.



6. Run your application using either the *F5* key or the **Run** | **Start Debugging** menu item.

There's more...

Be careful while connecting the USB cable to your Vita. For some unfathomable reason, you can easily put it in upside down! If you aren't getting a connection to your device, be sure to check if your cable is in upside down mode! I thought my Vita was broken the first time I encountered this, as I left it charging, or at least I thought I did. When I came back and it was completely dead, it took a few minutes of head-scratching until I figured out what was wrong.

See also

- The example locations can be found in the *Installing the PlayStation Mobile SDK* recipe
- See the *Deploying to PlayStation Certified Android Devices* recipe for details on managing App Keys using PublishingUtility

Manipulating an image dynamically

This recipe will demonstrate using the imaging SDK to resize and crop an image, and then saving that image to the gallery.

Getting ready

This recipe builds on the example created in the prior two recipes. The complete source code is available in `Ch01_Example4`.

How to do it...

In the `Initialize()` method, change the code as follows:

```
Image image = new
Image(ImageMode.Rgba,new
ImageSize(500,300),new
ImageColor(0,0,0,0));
Image resizedImage, croppedImage;
image.DrawText("Hello World",
   new ImageColor(255,255,255,255),
   new
Font(FontAlias.System,96,FontStyle.Italic),
   new ImagePosition(0,150));
croppedImage = image.Crop (new
ImageRect(0,0,250,300));
resizedImage =
croppedImage.Resize(new
ImageSize(500,300));
_texture = new
Texture2D(resizedImage.Size.Width,resizedImage.Si
_texture.SetPixels(0,resizedImage.ToBuffer());
resizedImage.Export("My
Images","HalfOfHelloWorld.png");
image.Dispose();
```

```
resizedImage.Dispose();
croppedImage.Dispose();
```

How it works...

First, we dynamically generate our image just like we did in the previous recipe. We then crop that image to half its width, by calling the `Crop()` function and providing an `ImageRect` variable half the size of the image. Crop returns a new image (it is not destructive to the source image) that we store in `croppedImage`. We then resize that image back to the original size by calling the `Resize()` function on `croppedImage`, with the original size specified as an `ImageSize` object. Like `Crop()`, `Resize()` is not destructive and returns a new image that we store in `resizedImage`. We then copy the pixels from `resizedImage` into our texture using `SetPixels()`.

Next, we call `Export()`, which saves our cropped and resized image in a folder called `My Images` as a file named `HalfOfHelloWorld.png` on your device. Finally, we call `Dispose()` on all three images, to free up the memory they consumed.

Now if you run your code, instead of "Hello World", you simply get "Hello".

There's more...

When run on the simulator, `export` will save the image to your `My Pictures` folder. Be sure to call `Dispose()`, or wrap within a `using` statement all objects that implement `IDisposable`, or you will quickly run out of memory. Most resources like images and textures require disposal.

Working with the filesystem

This recipe illustrates the various ways you can access the filesystem.

Getting ready

The complete code for this example is available in `Ch1_Example5`. This example adds the following two new `using` statements:

- `using System.Linq;`
- `using System.IO;`

How to do it...

Create a new solution and enter the following code replacing the `Main()` function:

```
public static void Main (string[]
args){
 const string DATA = "This is some
data";

 byte[] persistentData =
PersistentMemory.Read();

 byte[] restoredData =
persistentData.Take(DATA.Length *
2).ToArray();
 string restoredString =
System.Text.Encoding.Unicode.GetString(restoredDa

 byte[] stringAsBytes =
System.Text.Encoding.Unicode.GetBytes(DATA);

 for(int i = 0; i <
stringAsBytes.Length; i++)
```

```
  persistentData[i] =
stringAsBytes[i];
 PersistentMemory.Write
(persistentData);

 using(FileStream fs = File.Open("/
Documents/
Demo.txt",FileMode.OpenOrCreate)){
  if(fs.Length == 0)

fs.Write(stringAsBytes,0,stringAsBytes.Length);
  else{
   byte[] fileContents = new
byte[fs.Length];
   fs.Read(fileContents,0,
(int)fs.Length);
  }

 }
}
```

How it works...

The first part of this sample demonstrates using the `PersistentMemory` class. It is a statically available class, so you cannot allocate one. Instead you access it using `PersistentMemory.Read()`. This returns a byte array that you can now modify. We read some data using the LINQ extension `Take()` method. We multiplied the size by 2 because the string is stored as UNICODE, which requires 2 bytes per character. We then convert those bytes into a Unicode string. Next, we convert our `DATA` string into a byte array, then write it to `persistentData` byte by byte in a `for` loop. Finally we commit our changes to `persistentMemory` by calling `PersistentMemory.Write()`, passing in our updated `persisitentMemory` byte array.

The next part of the recipe demonstrates traditional file access using the standard `.NET` libraries, in this case `File` and `FileStream`. We open a text file in the `/Documents` folder named `Demo.txt`, creating it if it doesn't exist. If the file doesn't exist, we write our byte

array `stringAsBytes` to the file. Otherwise, we read the file contents into a new byte array `fileContents`.

The first time you run this example, `PersistentMemory.Read()` will contain gibberish, as nothing has been written to it yet. On the second run, it will start with the bytes composing your `DATA` string.

There's more...

Persistent storage is limited to 64 KB in space. It is meant for quickly storing information, such as configuration settings. It is physically stored on the filesystem in the `/save` directory in a file named `pm.dat`.

There are three primary locations for storing files:

- `/Application`: This is where your application files reside, as well as files added with the "Content" build action. These files are read only.
- `/Documents`: This is where you put files that require read and write access.
- `/Temp`: This is where you put temporary files, that can be read and written. These files will be erased when your program exits.

The PlayStation Mobile SDK limits directory structures to at most 5 levels deep, including the filename. Therefore, `/documents/1/2/3/myfile.txt` is permitted, but `/documents/1/2/3/4/myfile.txt` is not.

> **Note**
>
> The `PersistentMemory` classes was deprecated in the 1.0 release of PSM SDK and should no longer be used. Use traditional `.NET` file methods instead.

See also

- See the *Loading, displaying, and translating a textured image* recipe for details about how to add files to your

app's folders using build actions

Handling system events

This recipe covers handling the `OnRestored` system event.

Getting ready

The complete code for this example is available in `Ch01_Example06`.

How to do it...

Replace `Main()` with the following code:

```
public class AppMain
{
 static bool _done = false;
 public static void Main (string[]
args){

  SystemEvents.OnRestored +=
HandleSystemEventsOnRestored;
  while(!_done)  {
   SystemEvents.CheckEvents();
   // Loop until application
minimized then restored.
  }
 }

 static void
HandleSystemEventsOnRestored (object
sender, RestoredEventArgs e)
 {
  Console.WriteLine ("System
restored, ok to shut down");
  _done = true;
 }
}
```

How it works...

This code starts by wiring an `OnRestored` event handler to global class `SystemEvents`. We then loop until the `_done` bool is set to true. Within our loop we poll `SystemEvents.CheckEvents()` to see if any events have occurred. If an `OnRestored` event occurs, our event handler will be fired.

Our event handler `HandeSystemEventsOnRestored()` simply writes out a message to the console, then sets the `_done` bool to true, causing our loop to end, and our program to exit.

Run this example, then minimize the simulator or change applications on your device. When you refocus the application, it will fire the `OnRestored` event, causing your program to exit.

Chapter 2. Controlling Your PlayStation Mobile Device

In this chapter we will cover:

- Handling the controller's d-pad and buttons
- Using the Input2 wrapper class
- Using the analog joysticks
- Handling touch events
- Using the motion sensors
- Creating onscreen controls for devices without gamepads
- Configuring an Android application to use onscreen controls

Introduction

In this chapter, we are going to look at the various ways you can control your PlayStation Mobile device. The PlayStation Vita includes touch sensors, a full d-pad, two analog sticks, motion sensors, and more. It has to stay compatible with other devices, so features such as the rear touch panel are not supported. Not all devices support all of the PS Vita's features; in fact, most do not. This is especially true regarding physical joystick support. Fortunately, Sony has provided a tool for making onscreen controls, which we will cover shortly. The onscreen controls are built directly into the PSM virtual machine; you simply configure the screen layout the way you wish to see it appear.

There are, however, some limitations, especially when it comes to what you can do in the simulator. For some of these recipes, you will require an actual physical device to test them. I will point out at the beginning of each recipe if there are special requirements.

You may also notice, examples in this chapter make use of the **GameEngine2D** framework. This is done to cut down the length of example source listings. As you may have seen from the recipes in

Chapter 1, *Getting Started*, without using the framework, accomplishing some rather simple tasks can take a fair bit of code. Do not worry; we will cover GameEngine2D in greater detail in future recipes.

Handling the controller's d-pad and buttons

In this recipe, we will look at how to read the device's gamepad's d-pad and button states. The code remains the same regardless of whether it is a physical device or if the controls are emulated.

Getting ready

Load up **PlayStation Mobile Studio** and create a new project. Add a reference to `Sce.PlayStation.GameEngine2D` and `Sce.PlayStation.GameEngine2D.Base`. The complete source for this example can be found in `Ch2_Example1`.

How to do it...

In the `AppMain.cs` file of your newly created project, enter the following code:

```
using System;
using System.Collections.Generic;

using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Environment;
using Sce.PlayStation.Core.Graphics;
using Sce.PlayStation.Core.Input;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Base;

namespace Ch2_Example1
{
  public class AppMain
  {
```

```
    public static void Main (string[]
args)
    {
      Director.Initialize();
      Scene scene = new Scene();

scene.Camera.SetViewFromViewport();
      Label label = new Label();
      label.Position = new
Vector2(0,Director.Instance.GL.Context.GetViewpor

      scene.AddChild(label);

Director.Instance.RunWithScene(scene,true);
      bool quitApp = false;

      while(!quitApp)
      {
        Director.Instance.Update();
        string currentStatus = "You
are currently pressing:\n";
        var pressedButtons =
GamePad.GetData(0).ButtonsDown;

        if((pressedButtons &
GamePadButtons.Up) ==
GamePadButtons.Up){
          currentStatus += "Up
button";
        }
        if((pressedButtons &
GamePadButtons.Left) ==
GamePadButtons.Left){
          currentStatus += "Left
button";
        }
        if((pressedButtons &
GamePadButtons.Right) ==
GamePadButtons.Right){
          currentStatus += "Right
```

```
button";
            }
            if((pressedButtons &
GamePadButtons.Down) ==
GamePadButtons.Down){
              currentStatus += "Down
button";
            }
            if((pressedButtons &
GamePadButtons.Cross) ==
GamePadButtons.Cross){
              currentStatus += "Cross
button";
              }


if((GamePad.GetData(0).ButtonsPrev &
GamePadButtons.Cross) ==
GamePadButtons.Cross){
                quitApp = true;
            }
            label.Text = currentStatus;
            Director.Instance.Render();

Director.Instance.GL.Context.SwapBuffers();
            Director.Instance.PostSwap();
          }
        }
      }
}
```

How it works...

The initial code is all about setting up the `GameEngine2D` classes, which we will cover in detail later in the book. Essentially we just set up our scene then create a new `Label` object, which we will use to display the current control status on the screen. We then start our scene running and loop until the bool `quitApp` is set to true.

In our loop, we have to make four required calls using the

`Director` object, which again we will cover later in this chapter. It is the remainder of the loop that is the focus of this recipe.

With each iteration through the loop we clear out our label's text. We then get the current state of our game pad using the following call:

```
var pressedButtons =
GamePad.GetData(0).ButtonsDown;
```

At this point `pressedButtons` contains a `GamePadButtons` enum representing all of the buttons that are currently down. Think of this like a bit field of Boolean values, with the corresponding bit for each button *on* if the button is currently pressed. Therefore, to test if a certain button is down, we perform a binary and (*&*) operation, testing it against the button value we want to check.

We check the status of the Up, Left, Right, Down, and Cross buttons and update our label if any of these buttons is currently down. As you can see, the d-pad is basically just a collection of four individual buttons that are either on or off. There are additional buttons to the ones demonstrated here, including the left and right shoulder buttons and the start button. You handle them in the exact same manner. An action very similar to assigning `pressedButtons` is performed when we make the following call:

```
if((GamePad.GetData(0).ButtonsPrev &
GamePadButtons.Cross) ==
GamePadButtons.Cross)
```

This works almost identical to when we checked the down button state, but instead it returns an enum representing buttons that were previously down. So if that button was pressed the prior time you checked input, it will be set. In this case, if the user previously pressed the `cross` button, we set `quitApp` to true, causing our program to exit.

The remaining code simply updates the label's text with our updated button status and makes calls to `Director` that are required if you

manually manage your GameEngine2D game loop.

There's more...

In addition to checking if a button is down using `ButtonsDown`, or if it was previously down using `ButtonsPrev`, there are also options for checking if a button was released this frame using `ButtonsUp`, as well as the general `Buttons` enum, which tells you the button's current status regardless of when it occurred.

The `GamePadData` struct that `GetData()` returns also contains a bool named `Skip`. This value is used to indicate if the input has already been handled elsewhere in code. This is useful as GamePad is globally available, so it is possible that you handle input already in a different part of your code. If so, you can set `Skip` to true, letting any code that calls `GamePad.GetData()` again before it is updated know that this input has already been handled elsewhere. The use of `Skip` is completely optional.

If you are working using the simulator, you may be wondering how exactly you press buttons that don't exist. The following table illustrates the key mappings when running the simulator:

Emulated button	Keyboard key
Left d-pad	Left arrow key
Right d-pad	Right arrow key
Up d-pad	Up arrow key
Down d-pad	Down arrow key
Square button	A
Triangle button	W
Circle button	D
Cross button	S
SELECT button	Z
START button	X
Left shoulder button	Q
Right shoulder button	E

Sadly, you cannot currently emulate analog sticks using the simulator, although an upcoming release intends to add Dual Shock

support.

> **Note**
>
> There is currently a bug in PSM Studio that prevents code completion from working for recently added references. You can work around this problem however. After you add a new library reference, save your project and exit PSM Sutdio. Once you restart PSM Studio, code completion should work normally. You can also simply add an additional library reference, then immediately remove it, and code completion will then be working properly. This prevents you from having to restart Studio.

See also

- See the *A game loop, GameEngine2D style* recipe in Chapter 3, *Graphics with GameEngine2D* for more details on how the `GameEngine2D` portions of this sample work

Using the Input2 wrapper class

In the previous recipe, you may have found the bit masking process a little crude. Apparently so did Sony, as they provided a helper method in the `GameEngine2D` library named `Input2` that makes dealing with input a bit smoother. In this recipe, we are going to demonstrate how that class works.

Getting ready

Open or duplicate the code you created in the previous recipe. Only the contents of the `while` loop are going to change in this recipe. The full code for this recipe is available at `Ch2_Example2`.

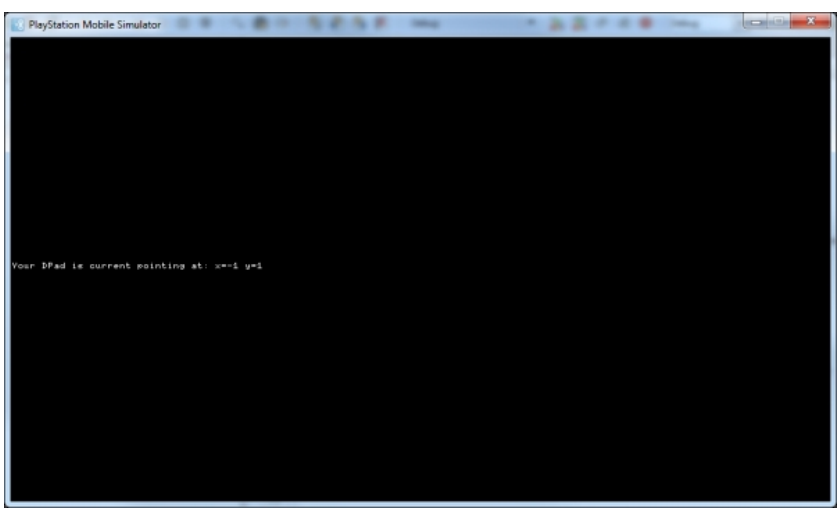
How to do it...

Open `AppMain.cs` and change the contents to the following code:

```
while(!quitApp)
      {
        Director.Instance.Update();
        string currentStatus = "Your
DPad is current pointing at: x=";

        currentStatus +=
Input2.GamePad0.Dpad.X;
        currentStatus += " y=";
        currentStatus +=
Input2.GamePad0.Dpad.Y;
        label.Text = currentStatus;

if(Input2.GamePad0.Cross.Press)
          quitApp = true;


if(Input2.GamePad0.Circle.Release)
          quitApp = true;
```

```
        if(Input2.GamePad0.Square.Down)
                quitApp = true;

        if(Input2.GamePad0.Triangle.On)
                quitApp = true;


                Director.Instance.Render();

        Director.Instance.GL.Context.SwapBuffers();
                Director.Instance.PostSwap();
              }
```

Now if you run the app, you will see the following output:



Move the d-pad in different directions and the *x* and *y* location will be displayed. You can press any face button to quit.

How it works...

This code simply gets the status of the d-pad and quits if either the `Cross` button is pressed, the `Circle` button is released, the

`Square` button is down, or the `Triangle` button is `On`. The rest of the code is the same boilerplate code you need to work with `GameEngine2D`. So what exactly is different here?

Well first off, the d-pad is no longer represented as four independent buttons. Instead it is represented as a normalized 2D vector indicating which direction it is pointing. Pressing the up and left arrow will give you a value of (-1,-1), pressing no buttons will give a value of (0,0), and pressing the down and right arrow will give you a value of (1,1). This allows you to treat the d-pad as a single entity instead of as four separate buttons.

Additionally, all of the various button states we saw in the earlier recipe are still there. Using `Input2` they are instead represented at a series of bool values, making the code slightly less compact, but certainly more readable. Keep in mind; this is basically just a helper layer over the top of the standard input functionality. The `Input2` helper class provides no new functionality or information, it is simply easier to use.

Finally `GamePad.GetInput(0)` has the equivalent shortcut reference `Input2.GamePad0`, because as of this point in time you will never have more than one controller.

There's more...

The `Input2` helper class is a static global variable. It is available for use once you have referenced `GameEngine2D.Base`.

You can also call `SetData()` and override the values returned by `GetData()`. This can be useful when you want to programmatically set controls, such as during testing, remapping controls dynamically or driving a replay. You simply set a `GamePadData` structure to the values you want and pass it into `SetData()`.

See also

- See the *Handling the controller's d-pad and buttons* recipe for the code this recipe is based on, as well as to see the way input is handled without the `Input2`

wrapper to help

Using the analog joysticks

In this recipe, we will demonstrate how to use the device's analog sticks, if they exist. If they do not exist, the onscreen controller will appear automatically.

Getting ready

In PSM Studio, create a new project and add a reference to `Sce.PlayStation.GameEngine2D` and `Sce.PlayStation.GameEngine2D.Base`. This recipe also requires you to add an image file to be displayed. I am bringing back our trusty F-18 graphic, but you can substitute whatever image you desire as long as it is in the proper format (`.png`, `.bmp`, or `.jpg`). The full code for this recipe is available at `Ch2_Example3`.

This example requires an actual device!

How to do it...

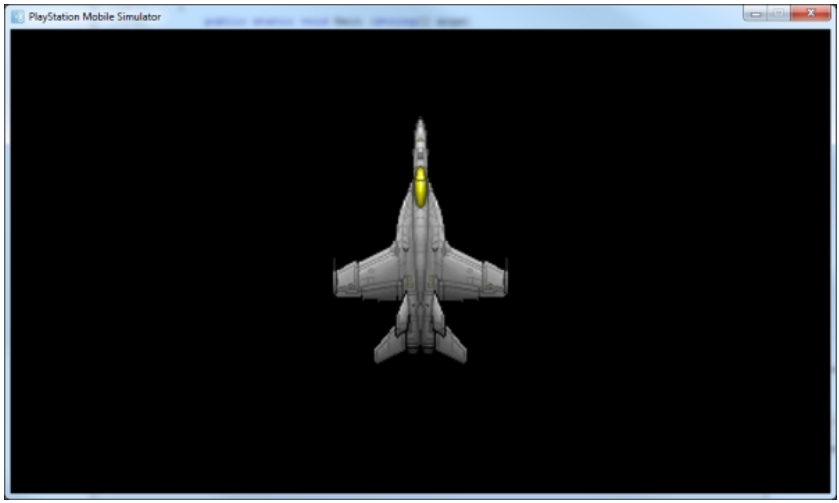
Open `AppMain.cs` and enter the following code:

```
using System;
using System.Collections.Generic;

using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Environment;
using Sce.PlayStation.Core.Graphics;
using Sce.PlayStation.Core.Input;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Base;

namespace Ch2_Example3
{
```

```
  public class AppMain
  {
    public static void Main (string[]
args)
    {
      Director.Initialize();
      Scene scene = new Scene();

scene.Camera.SetViewFromViewport();
      TextureInfo ti = new
TextureInfo("/Application/
FA-18H.png");
      SpriteUV sprite = new
SpriteUV(ti);
      sprite.Quad.S =
ti.TextureSizef;
      sprite.Position = new Vector2(

Director.Instance.GL.Context.GetViewport().Width/
- sprite.Quad.Center.X,

Director.Instance.GL.Context.GetViewport().Height
- sprite.Quad.Center.Y);

      scene.AddChild(sprite);

Director.Instance.RunWithScene(scene,true);
      bool quitApp = false;

      while(!quitApp)
      {
        Director.Instance.Update();


if(GamePad.GetData(0).AnalogLeftX <
0)
        {
          if(sprite.Position.X > 0)
            sprite.Position = new
Vector2(sprite.Position.X + 10 *
```

```
GamePad.GetData(0).AnalogLeftX
,sprite.Position.Y);
        }

if(GamePad.GetData(0).AnalogLeftX >
0)
        {
          if(sprite.Position.X <
Director.Instance.GL.Context.GetViewport().Width
- sprite.CalcSizeInPixels().X)
            sprite.Position = new
Vector2(sprite.Position.X + 10 *
GamePad.GetData(0).AnalogLeftX
,sprite.Position.Y);
        }
        Director.Instance.Render();

Director.Instance.GL.Context.SwapBuffers();
        Director.Instance.PostSwap();
      }
      ti.Dispose();
      Director.Terminate();
    }
  }
}
```

Here is the application running in the following output screen:



Move the left analog stick left and right to move the sprite; the amount you move the stick will determine the speed.

How it works...

Most of the initial code relates to `GameEngine2D` and will be covered in more detail in a later series of recipes. Basically, it sets up the `Director` singleton and `Scene` objects required for GE 2D, then loads our trusty `FA-18H.png` image into a `TextureInfo` object, which is then used to create our `SpriteUV` object, `sprite`. Next we size the `sprite` object to the same dimensions as its texture and then position it in the middle of the screen. Finally, we add our sprite to the scene and kick off our scene running with a call to `Director.Instance.RunWithScene()`.

Note

What's the difference between a sprite and an image?

People often use the term sprite and image interchangeably, but this isn't completely accurate. A sprite is a collection of pixels that can be drawn on the screen or elsewhere, such as a player graphic or score information. A sprite is often loaded from an image, but can also be generated dynamically. Additionally,

although a sprite can be loaded from an image, it is also possible to store a number of sprites within a single image. Therefore, a sprite is often an image, but doesn't have to be, while an image can contain multiple sprites.

It is the highlighted code within the `while` loop that we are most interested in. Just as in the previous recipe we read the Gamepad data using `GamePad.GetData(0)`, this time we are interested in the x axis of the left-hand stick, stored in `GamePad.GetData(0).AnalogLeftX`. This value returns a float with a value ranging from *-1.0* to *1.0*. A value of *-1.0* means the stick is being pressed all the way to the left, while a value of *+1.0* indicates the stick is being held all the way to the right. A value of *0.0* means the stick isn't being pressed at all.

Each pass through the loop we check to see if the left stick is being held to the left; if it is, we check to make sure our sprite isn't already all the way to the left, and if it isn't, we move it further to the left with the following call:

```
sprite.Position = new
Vector2(sprite.Position.X + 10 *
GamePad.GetData(0).AnalogLeftX
,sprite.Position.Y);
```

This works by updating the position to a new vector based on the current position `+ 10 *` pixels, the amount the stick is pressed to the left. So for example, if the stick is pressed halfway to the left, `AnalogLeftX` will be *-0.5* resulting in a value of *-5*, moving our position *5* pixels to the left. If the stick was pressed all the way to the left, the values would have been *-1.0 * 10 = -10*, which would move the position *10* pixels left. Unlike the d-pad, which is either on or off, the analog stick allows us to have a gradual range of values.

We then perform the same test, this time checking if `AnalogLeftX` is greater than *0*, meaning that the stick is being pressed to the right. In this case, we update the position to move to the right by a value ranging from *0* to *10.0* pixels, depending on how hard the stick is being pressed.

The remaining code is required when you manually manage a `GameEngine2D` loop. Finally, after our loop ends we behave like good citizens and clean up after ourselves, disposing of our `TextureInfo` object and calling `Terminate` on the static `Director` class.

There's more...

Although this recipe only utilized `AnalogLeftX` to read the left/right movement of the analog stick, there is also the value `AnalogLeftY` which holds vertical movement of the stick, with a value of *-1.0* representing a stick being pushed up, while a value of *1.0* means the stick is being held fully down. Additionally, the values `AnalogRightX` and `AnalogRightY` exist and hold the values of the right analog stick.

The `Input2` class also has a simplified representation for the two analog sticks. Instead of a pair of separate floats each, it represents the stick's over-all position using a single `Vector2` each, with (-1.0,-1.0) meaning the stick was being pushed to the top-left position, (0.0,0.0) representing a still stick, and a value of (1.0,1.0) indicating the stick was being pushed to the bottom-right corner. Obviously those represent the maximum ranges in each direction.

> **Note**
>
> Often when dealing with analog controls, it is normal to set a *dead zone*. This treats values close to (0,0) as basically being zero. This keeps stationary controls from being overly twitchy to extremely small motions. The PlayStation Mobile SDK automatically applies a dead-zone to analog controls, so you do not need to.

See also

- See the *Loading, displaying, and translating a textured image* recipe in Chapter 1, *Getting Started* for the download location of the sprite graphic used in this recipe

- See the *Using the Input2 wrapper class* recipe for more details on using the `Input2` helper class

Handling touch events

In this recipe we will look into how to deal with touch events.

Getting ready

In PSM Studio, create a new project and add a reference to `Sce.PlayStation.GameEngine2D` and `Sce.PlayStation.GameEngine2D.Base`. Then add an image file to your project (I will re-use the trusty `F18`) and make sure its **Build Action** is set to **Content**. The full code for this recipe is available as `Ch2_Example4`.

This example will run in the simulator; however, it does not currently support multitouch. To see multiple touches handled, you will need to run this code on an actual device.

How to do it...

Open `AppMain.cs` and enter the following code:

```
using System;
using System.Collections.Generic;

using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Environment;
using Sce.PlayStation.Core.Graphics;
using Sce.PlayStation.Core.Input;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Base;

namespace Ch2_Example4
{
  public class AppMain
```

```
  {
    public static void Main (string[]
args)
    {
      Director.Initialize();
      Scene scene = new Scene();

scene.Camera.SetViewFromViewport();

      TextureInfo ti = new
TextureInfo("/Application/
FA-18H.png");
      SpriteUV [] sprites = new
SpriteUV[6];
      sprites[0] = new SpriteUV(ti);
      sprites[1] = new SpriteUV(ti);
      sprites[2] = new SpriteUV(ti);
      sprites[3] = new SpriteUV(ti);
      sprites[4] = new SpriteUV(ti);
      sprites[5] = new SpriteUV(ti);
      sprites[0].Quad.S =
sprites[1].Quad.S = sprites[2].Quad.S
= sprites[3].Quad.S =
sprites[4].Quad.S
        = sprites[5].Quad.S =
ti.TextureSizef;


      float screenWidth =
Director.Instance.GL.Context.GetViewport().Width;
      float screenHeight =
Director.Instance.GL.Context.GetViewport().Height

      foreach(var sprite in sprites)
        scene.AddChild(sprite);



Director.Instance.RunWithScene(scene,true);
      bool quitApp = false;
```

```
        while(!quitApp)
        {
          Director.Instance.Update();
          foreach(SpriteUV sprite in
sprites)
            sprite.Visible = false;

          var touches =
Touch.GetData(0);
          for(int i = 0; i <
touches.Count;i++)
          {
            // Top Left: -.5, -.5
            // Top Right: .5, -5
            // Bottom Left: -.5, .5
            // Bottom Right: .5, .5
            sprites[i].Visible = true;
            sprites[i].Position = new
Vector2(
              (touches[i].X + 0.5f) *
screenWidth,
              screenHeight -
((touches[i].Y + 0.5f) *
screenHeight));
          }
          Director.Instance.Render();

Director.Instance.GL.Context.SwapBuffers();
          Director.Instance.PostSwap();
        }
        ti.Dispose();
        Director.Terminate();
      }
    }
}
```

How it works...

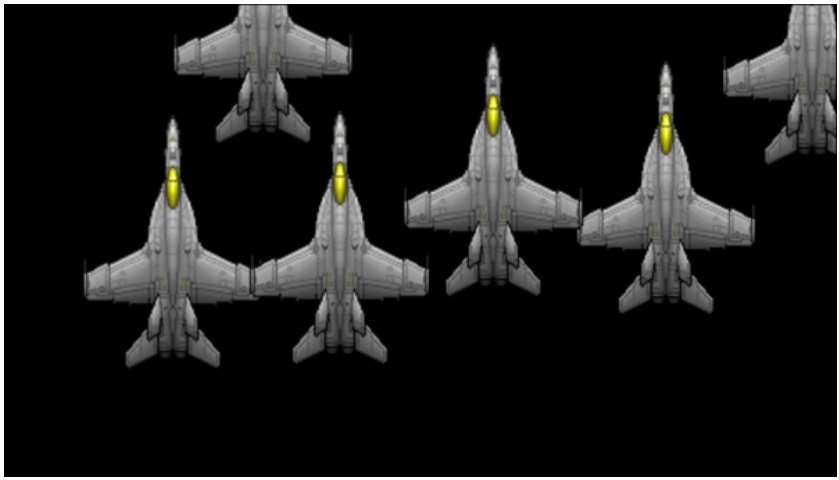
This example starts off with boilerplate `GameEngine2D` initialization

code. We then create our `TextureInfo` object to store our `F18` texture fire. We then create an array of six `SpriteUV` objects, all reusing the same texture. Next we size all six sprites equal to their source texture's dimensions. For convenience later on, we store the screen width and height. Next up, we add all six sprites to our scene, then kick it off with a call to `RunWithScene()`. Now we get to the heart of demonstration within the `while` loop.

With each iteration of the loop, in addition to the standard four required `Director` function calls, we do the following:

1. Loop through each of our `SpriteUV` object's in the `sprites` array, making them invisible.
2. Get the current touch data with a call to `Touch.GetData(0)`.
3. Loop through each of the touches in the list; for each location touched on the screen, we make a sprite visible at that position for this frame.
4. Finally we clean up after ourselves by disposing of our `TextureInfo` object and terminate our `Director` singleton.

Now if you run the application, wherever you touch the screen a sprite will appear as long as you are touching the screen. Move your finger and the sprites will move, remove your finger and the sprite will disappear. Here is the application running with six touches occurring:



There's more...

The following line of code most likely jumped out at you:

```
sprites[i].Position = new Vector2(
(touches[i].X + 0.5f) * screenWidth,
      screenHeight - ((touches[i].Y +
0.5f) * screenHeight));
```

What exactly are we doing here? The touch data *x* and *y* value returned in `TouchData` are screen coordinates ranging in value from *-0.5* to *+0.5*. Therefore, we simply add *0.5* to the coordinate to make it a value from *0.0* to *1.0*, then multiply it by our screen dimensions to give actual screen pixel coordinates. We then set the sprite's position to be at the pixel location the user is currently touching. Why then return a value between *-0.5* and *+0.5* instead of returning a value ranging from *0.0* to *1.0*? That's a very good question! I cannot think of a good reason.

Another important thing to be aware of is, different devices may support a different maximum number of simultaneous touches. The PlayStation Vita tracks a maximum of six touches. Each touch is also assigned an ID value; however, if you have a situation where two fingers touch multiple times, how this is handled is device

dependent. Some devices might report this as 1, 2, 1, 2 while other devices will report 1, 2, 3, 4. All concurrent touches though will be assigned an individual ID. In addition to checking if a touch is down, `TouchStatus` also indicates up, move, and cancelled statuses.

Just as when dealing with gamepads, touch data is also provided by the `Input2` wrapper class. It represents touch statuses as a series of bools, as well as returning the touch location as much more sensible `Vector2` coordinates in normalized screen coordinates ranging from *-1.0* to *1.0*.

Currently, the PSM SDK does not provide programmatic access to the rear touchscreen.

> **Note**
>
> The preceding image was taken as a screenshot on my PS Vita. If you want to take a screenshot, you simply press and hold the PS button and Start button at the same time. The screen will flash and your screenshot will be saved. You can access it in the Gallery section of the Photo application. You may be asking, *How exactly do you touch the screen in six different locations, while simultaneously holding the PS Button and Start button*? The answer... with great difficulty!

See also

- See the *Using the Input2 wrapper class* recipe for more details on using the `Input2` helper class

Using the motion sensors

In this recipe we will look into how to use the accelerometer and gyroscope.

Getting ready

Open PSM Studio and create a new project, adding references for `GameEngine2D` and `GameEngine2D.Base`. The complete code for this recipe is available at `Ch2_Example5`.

This example will not run in the simulator. Some Android based devices may not support the gyro, so `AngularVelocity` may not be available.

How to do it...

Open `AppMain.cs` and enter the following code:

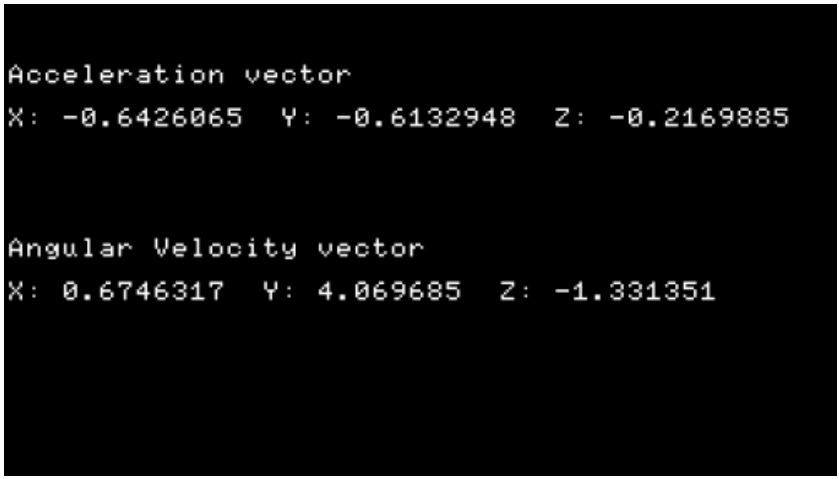
```
using System;
using System.Collections.Generic;

using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Environment;
using Sce.PlayStation.Core.Graphics;
using Sce.PlayStation.Core.Input;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Base;

namespace Ch2_Example5
{
  public class AppMain
  {
    public static void Main (string[]
```

```
args)
    {
      Director.Initialize();
      Scene scene = new Scene();

scene.Camera.SetViewFromViewport();

      float screenWidth =
Director.Instance.GL.Context.GetViewport().Width;
      float screenHeight =
Director.Instance.GL.Context.GetViewport().Height

      Label labelAcceleration = new
Label();
      labelAcceleration.Text =
"Acceleration vector";
      labelAcceleration.Position =
new Vector2(0,450);
      labelAcceleration.Scale = new
Vector2(3,3);

      Label labelAngularVelocity =
new Label();
      labelAngularVelocity.Text =
"Angular Velocity vector";
      labelAngularVelocity.Position =
new Vector2(0,250);
      labelAngularVelocity.Scale =
new Vector2(3,3);

      Label labelPosition = new
Label();
      labelPosition.Position = new
Vector2(0,350);
      labelPosition.Scale = new
Vector2(3,3);

      Label acceleration = new
Label();
      acceleration.Position = new
```

```
Vector2(0,400);
      acceleration.Scale = new
Vector2(3,3);

      Label angularVelocity = new
Label();
      angularVelocity.Position = new
Vector2(0,200);
      angularVelocity.Scale = new
Vector2(3,3);

scene.AddChild(labelAcceleration);
      scene.AddChild(acceleration);
      scene.AddChild(labelPosition);

scene.AddChild(labelAngularVelocity);

scene.AddChild(angularVelocity);


Director.Instance.RunWithScene(scene,true);
      bool quitApp = false;


      while(!quitApp)
      {
        Director.Instance.Update();

        var motionData =
Motion.GetData(0);

        acceleration.Text =
          "X: " +
motionData.Acceleration.X.ToString()
+
          "  Y: " +
motionData.Acceleration.Y.ToString()
+
          "  Z: " +
motionData.Acceleration.Z.ToString();
```

```
            labelPosition.Text = "";

        if(System.Math.Round(motionData.Acceleration.Z,1)
        == -1)
              labelPosition.Text =
        "Facing up";

        if(System.Math.Round(motionData.Acceleration.Z,1)
        == 1)
              labelPosition.Text =
        "Facing down";
              else
        if(System.Math.Round(motionData.Acceleration.Y,1)
        == -1)
              labelPosition.Text =
        "Facing you";
            angularVelocity.Text =
              "X: " +
        motionData.AngularVelocity.X +
              "  Y: " +
        motionData.AngularVelocity.Y +
              "  Z: " +
        motionData.AngularVelocity.Z;

            if(Input2.GamePad0.Cross.Down
        == true)
              quitApp = true;
            Director.Instance.Render();

        Director.Instance.GL.Context.SwapBuffers();
            Director.Instance.PostSwap();
          }
          Director.Terminate();
        }
      }
    }
```

Here is the application running:



As you move the device around, the values will update depending on the speed and position. Press the **cross** button to exit.

How it works...

The first half of this code is the boilerplate `GameEngine2D` code, which is probably becoming rather familiar at this point. Additionally, we create four text labels for displaying our motion data on screen, which we then add to our scene object and run using the `Director` function. The only somewhat confusing call in the mix, if you have read the prior recipes in this chapter, is most likely the `Scale = new Vector2(3,3);` function. This simply scales the label by a factor of `3`, as the default is so small as to be almost unreadable, at least to my aging eyes.

As before, the bulk of the new functionality is within the `while` loop. Once again, we have the typical four `Director` method calls. What we are doing here is:

- Getting the motion data and storing it in a `motionData` variable with a call to the static global `Motion.GetData()` object.
- Assigning the acceleration label the `X`, `Y`, and `Z` acceleration values within the `motionData` variable.
- Clearing out the `labelPosition` label text, then assigning it the value `"Facing up"` if the z axis is *-1*,

`"Facing down"` if the z axis is *1*, or assigning the value `"Facing you"` if the y axis is equal to *1*. All of these values are rounded off to the nearest decimal place.

- We then assign the `X`, `Y`, and `Z` values of `AngularVelocity` in `motionData` to the `angularVelocity` label.
- Finally, we check to see if the **Cross** button is down, if it is we exit the application.

Now when you run the code and it will constantly update with the values from the accelerator and internal gyro (if your device has one).

There's more...

The naming decisions are rather odd and a bit confusing when it comes to motion data. Acceleration is probably easier thought of as the direction your device is facing in 3D space, while AngularVelocity describes its motion along each axis.

When trying to make sense of direction coordinates, think of it this way. Relative to the screen of the device, if the device is facing up (as in, lying flat on a table with the screen facing the sky), it will have an acceleration value of approximately (0,0,-1). If you flip the device over to face down, it will have a value of (0,0,1). While if you pick up the device so that the screen faces you with the cable socket facing the ground, it will have a value of (0,1,0). Run this sample program to get an idea of other positional values, as well as to see the effect of moving the device on the `AngularVelocity` vector.

See also

- See the *Using the Input2 wrapper class* recipe for more details on using `Input2` helper class

Creating onscreen controls for devices without gamepads

In this recipe, we will look at creating onscreen controls for devices that do not have physical controls. We are going to configure a control set that has a left hand analog stick, select, start, as well as the cross, and triangle buttons. We are going to layout controls for an example device, the *Xperia Arc*, which has a 4.2" 480 x 854 screen.

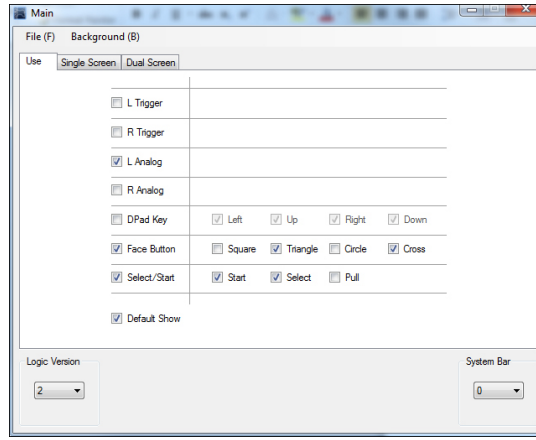
Getting ready

This recipe can be created on your computer, however you will require an Android device to test it, as onscreen controls cannot be run on the Vita or simulator.

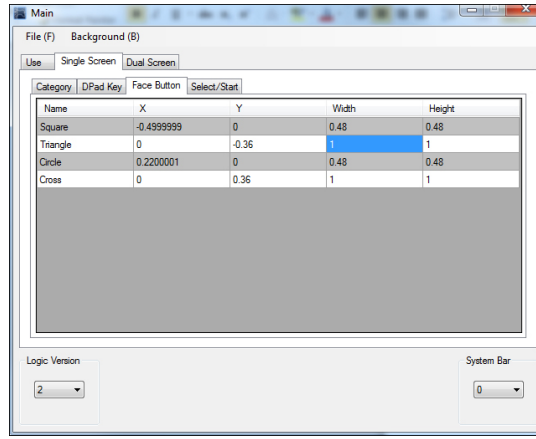
To get started, locate and execute `OscCustomizeTool.exe`. It will be located in the `Tools` folder, under the PSM Studio `install` directory. On my machine, which was installed using defaults, this file was located at `C:\Program Files (x86)\SCE\PSM\tools\OscCustomizeTool`.

How to do it…

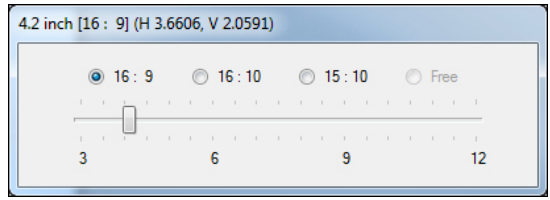
1. On the main screen, uncheck all buttons except the **L Analog**, **Select/Start**, **Select**, **Start**, **Face Button**, **Cross**, and **Triangle** buttons like this:



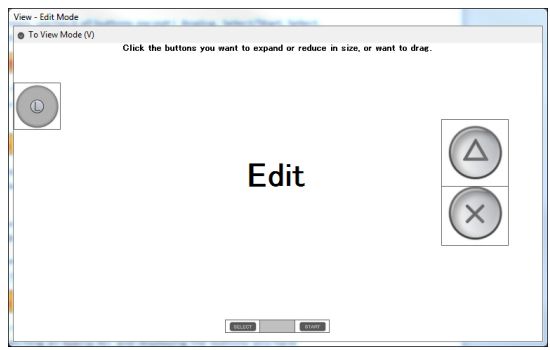
2. Switch over to the **Single Screen** tab, change the **Face Button** button's **Height** to `2`, then select the **Face Button** sub-tab and resize **Triangle** and **Cross** to have a width and height of **1**.



3. You can now optionally add a background image that will be displayed under your controls. Select **Background** | **Load** and select an image to be used as a background.
4. In the layout window, select the checkbox next to **To View Mode** (once checked, it will display **To Edit Mode**).
5. Now switch over to the sizing window, select a **16:9** aspect ratio, and drag the slider to `4.2` (the size is displayed in the title of the window).



6. Switch back to the layout window, which should now be resized to the dimensions matching an Xperia Arc phone and displaying the buttons you have selected. Switch back to **Edit** mode. Now drag the buttons around, where you want them to be displayed.



7. Return to the main window and select the menu **File** | **Save**; save the `osc.cfg` file for later use.

How it works...

The `OscCustomizeTool` ultimately produces the `osc.cfg` file, which is read by the device to create the onscreen controls. It is important that you do not change the resulting filename.

The layout window is for visualization purpose only and doesn't actually change the positioning of the controls. The same layout is used by devices of all size, so you simply modify the values to see how your control design will look on different sized screens.

There's more...

The following table represents the typical resolutions and aspect

ratios of supported PlayStation Mobile Android devices:

Device ratio (inches) and resolution		
[illegible]		
[illegible]		
[illegible]		
[illegible]		
[illegible]		
[illegible]		
[illegible]		
[illegible]		
[illegible]		

The value **System Bar** in the main window determines how much space is left for the Android controls. For some devices, such as many Android 4 tablets, a portion of space needs to be reserved for the system controls such as the back and home buttons.

The system will automatically provide an onscreen controller for you. You only need to go through this process if you want to provide something different than the default gamepad. There are limitations to how much you can modify the controls as well to keep things somewhat consistent. Items on the left-hand side of the screen cannot be moved to the right-hand side and vice versa.

See also

- See the *Configuring an Android application to use onscreen controls* recipe for details on how to actually deploy your control settings to a device

Configuring an Android application to use onscreen controls

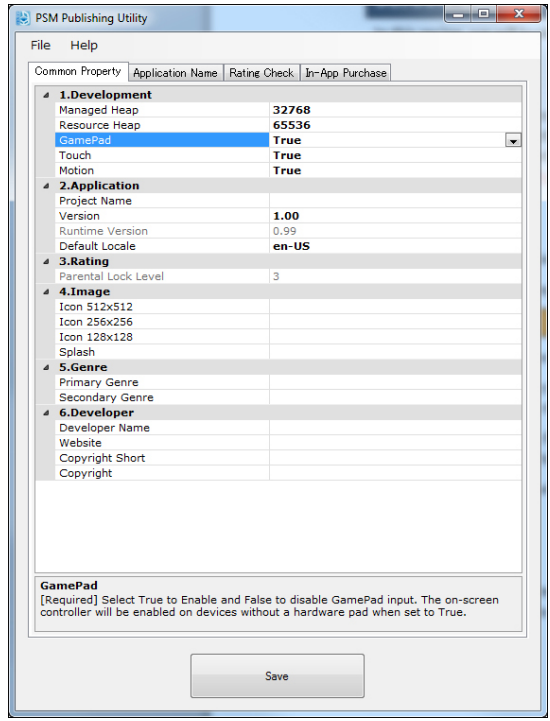
In this recipe, we will look at configuring onscreen controls on an Android device.

Getting ready

You need to have run through the prior recipe, or generated the `osc.cfg` file, which needs to be named exactly that. Locate the application `PublishingUtility.exe`. On my default install, it is located at `C:\Program Files (x86)\SCE\PSM\tools\PublishingUtility`. You will also need a PSM project you want to add the controls to.

How to do it…

1. Load `PublishingUtility.exe`.
2. Locate the Gamepad section and, in the drop-down to the right, set it to `true`.



3. Click on the **Save** button and save the file as `app.xml` in the same file location as `osc.cfg`.
4. In the project you want to add onscreen controls to, add `app.xml`, allowing it to overwrite the default.
5. Also add the file `osc.cfg`, then right-click it in the **Solution** view and set its **Build Action** to **Content**.

You are now complete; when the project is run on an Android device the customized onscreen keyboard will be shown.

How it works...

PublishingUtility generates the `app.xml` file for you, just as the `OscCustomizeTool` generated the `osc.cfg` file. You could have edited the `app.xml` file manually instead of loading the publishing tool. One is created for you by default when you create a new project.

There's more...

As you probably noticed, PublishingUtility handles a lot more than just setting Android configuration settings. We will look at the utility in greater detail in a later chapter.

See also

- See the *Creating onscreen controls for devices without gamepads* recipe for details on how to generate the required `osc.cfg` file

Chapter 3. Graphics with GameEngine2D

In this chapter we will cover:

- A game loop, GameEngine2D style
- Creating scenes
- Adding a sprite to a scene
- Creating a sprite sheet
- Using a sprite sheet in code
- Batching a sprite with SpriteLists
- Manipulating a texture's pixels
- Creating a 2D particle system

Introduction

At its core, PlayStation Mobile is actually a fairly low level SDK, just a small step above OpenGL. As you may have seen in Chapter 1, *Getting Started*, this results in a rather large amount of code to create 2D graphics, as you are working in 3D. Fortunately, Sony has provided a 2D library for you, GameEngine2D.

If you have ever used the popular Cocos2D library, GameEngine2D will feel immediately familiar, as Cocos2D was the inspiration behind GameEngine. GameEngine2D makes it significantly easier to make a 2D game.

A game loop, GameEngine2D style

In this recipe, we are going to create a game loop using the `GameEngine2D` library. The game loop is essentially the heart of your game, where input is read, graphics are drawn, and the game is updated. The `GameEngine2D` library helps you by making all of these aspects easier.

Getting ready

Load up PlayStation Mobile Studio and create a new project. Add a reference to `Sce.PlayStation.GameEngine2D`. The complete source for this example can be found in `Ch3_Example1`.

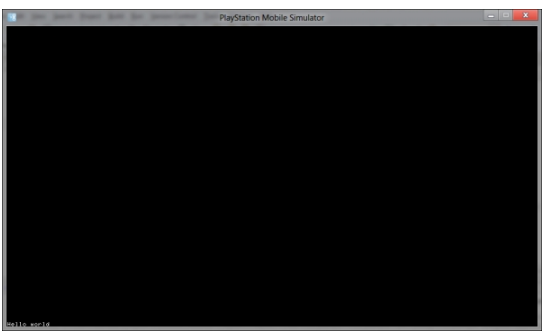
How to do it...

1. In the `AppMain.cs` enter the following code:

```
using System;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba


namespace Ch3_Example1
{
  public class AppMain  {
    public static void
Main(){
      Director.Initialize();
      Scene scene = new
Scene();

scene.Camera.SetViewFromViewport();
```

```
            var label = new
Sce.PlayStation.HighLevel.GameEngine2D.La
            label.Text = "Hello
world";

            scene.AddChild(label);

Director.Instance.RunWithScene(scene);
            Director.Terminate();
        }
    }
}
```

2. Finally hit the *F5* key to run our application:



How it works...

This is about the smallest amount of code you can write that actually still accomplishes something. This is the iconic "Hello World" example, implemented using the `GameEngine2D` library.

At the heart of the `GameEngine2D` library is the `Director` singleton, which needs to be initialized manually. Behind the scenes, the `Director` object manages the scenes, rendering and updating the `Scheduler` and `ActionManager`, two key subsystems we will cover shortly. The `Director` object needs to be initialized before you use any `GameEngine2D` objects.

Next, we create a `scene` object, then set the scene's camera to render using the fullscreen dimensions with the

`scene.Camera.SetViewFromViewport()` call. We then create a `Label` object, set its text to "Hello World", then add the label to our scene. Next, we call the `Director` singleton using its `Instance` member, calling `RunWithScene()` and passing in our newly created scene, causing our game to start. Finally, we call the `Terminate()` method that will clean up everything once the scene has finished running.

Note

Why was "Hello World" printed in the bottom-left corner?

GameEngine2D is based on Cocos2D and Cocos2D sets the origin point (0,0) at the bottom-left corner, instead of the more traditional top left corner. Ultimately, Cocos2D most likely used the bottom-left corner as the origin point because this is the coordinate system that OpenGL uses. When you add a node to a scene without specifying a position, it will be added at the origin. Therefore, when we added our label to the scene, it was located at the bottom-left corner.

There's more...

At this point you may be wondering, *How is this a game loop, there is no loop?* which is a very good question. There is still a loop, but it is hidden deep inside the `Director` class. We will be covering this in a bit more detail, in a later chapter. You can, however, have more control over the process by implementing your own game loop manually. If you want to manually control the main game loop (which Sony recommends), pass in `true` as the second parameter in the `RunWithScene` call, like this:

```
Director.Instance.RunWithScene(scene,true);
```

Once you inform `Director` that you are going to manually handle the game loop, there are a series of calls you need to make in the proper order. The following is the minimum code required in a manually handled game loop:

```
Director.Instance.RunWithScene(scene,true);
while(true)
{
 Director.Instance.Update();
 Director.Instance.Render ();

Director.Instance.GL.Context.SwapBuffers();
 Director.Instance.PostSwap();
}
Director.Terminate();
```

Note

GameEngine2D source included

Sony has included the complete source code for GameEngine2D (as well as all of the other libraries in the `Sce.PlayStation.HighLevel` namespace) in the source subdirectory on your PlayStation Mobile SDK install. On my PC, installed using the defaults, this folder is located at `C:\Program Files (x86)\SCE\PSM\source`.

See also

- See the *Creating a simple game loop* recipe in Chapter 1, *Getting Started* for an example of a non-GameEngine2Dgame loop

Creating scenes

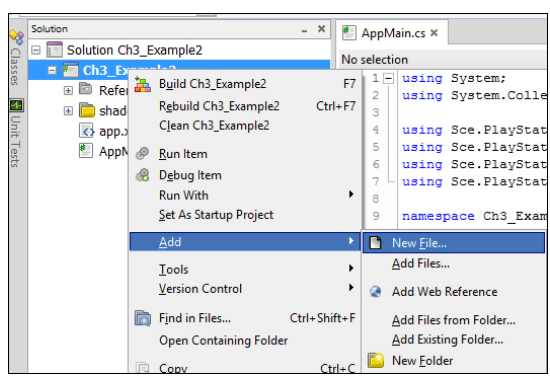
In this recipe, we are going to implement a derived scene object, `MyScene`.

Getting ready

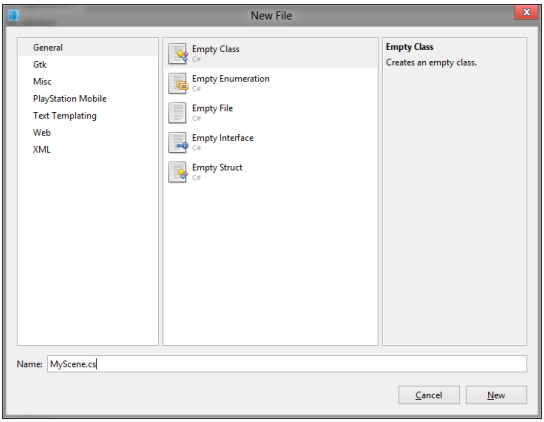
Load up PlayStation Mobile Studio and create a new project or duplicate the code from the *A game loop, GameEngine2D style* recipe, as the `AppMain.cs` code is going to be virtually unchanged. The complete source for this example can be found in `Ch3_Example2`.

How to do it...

1. In the **Solution** pane of PlayStation Mobile Studio, right-click on your Project (`Ch3_Example2`) and select the menu **Add** | **New File...**.



2. In the **New File** dialog box, select **General** on the left-hand side, **Empty Class** on the right, then at the bottom in the **Name:** field, enter `MyScene.cs`, and click on the **New** button.



3. In `MyScene.cs`, enter the following code:

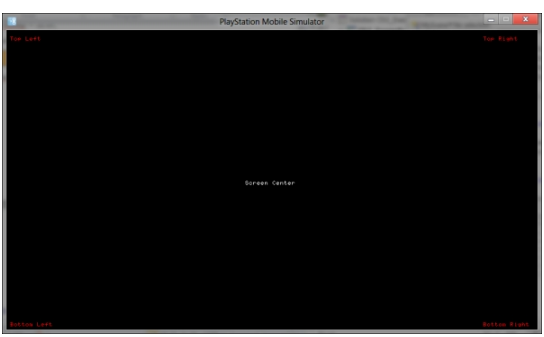
```
using System;
using Sce.PlayStation.Core;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba

namespace Ch3_Example2
{
  public class MyScene :
Scene
  {
    private Label
_labelTopLeft;
    private Label
_labelTopRight;
    private Label
_labelBottomLeft;
    private Label
_labelBottomRight;
    float elapsedTime = 0;
    public MyScene ()
    {

this.Camera.SetViewFromViewport();
      var width =
```

```
Director.Instance.GL.Context.Screen.Width
      var height =
Director.Instance.GL.Context.Screen.Heigh

      _labelTopLeft = new
Label() { Text="Top Left",
Position=new
Vector2(5,height-20) };
      _labelTopRight = new
Label() { Text="Top Right",
Position=new
Vector2(width-100,height-20)
};
      _labelBottomLeft = new
Label() { Text="Bottom
Left", Position=new
Vector2(5,5) };
      _labelBottomRight =
new Label() { Text="Bottom
Right", Position=new
Vector2(width-100,5) };


this.AddChild(_labelTopLeft);

this.AddChild(_labelTopRight);

this.AddChild(_labelBottomLeft);

this.AddChild(_labelBottomRight);


Scheduler.Instance.ScheduleUpdateForTarge
    }

    public override void
Update(float dt)
    {
      elapsedTime += dt;
      base.Update(dt);
```

```
    }

    public override void
Draw ()
    {
      var tenSecondsElapsed
=
(System.Math.Floor(elapsedTime/
10))% 2;
      if(tenSecondsElapsed
==0)

_labelBottomLeft.Color =
_labelBottomRight.Color =
_labelTopLeft.Color =
_labelTopRight.Color = new
Vector4(255,0,0,255);
      else

_labelBottomLeft.Color =
_labelBottomRight.Color =
_labelTopLeft.Color =
_labelTopRight.Color = new
Vector4(0,255,0,255);
      base.Draw ();
    }
  }
}

In AppMain.cs, replace
Main() with (changed
bolded):
public static void Main(){
      Director.Initialize();
      MyScene scene = new
MyScene();

scene.Camera.SetViewFromViewport();

      var label = new
```

```
Sce.PlayStation.HighLevel.GameEngine2D.La
      label.Text = "Screen
Center";
      label.Position = new
Sce.PlayStation.Core.Vector2(

Director.Instance.GL.Context.Screen.Width

Director.Instance.GL.Context.Screen.Heigh

      scene.AddChild(label);


Director.Instance.RunWithScene(scene);
      Director.Terminate();
    }
```

4. Run your application by hitting the *F5* key:



This application draws a label in each of the four corners, then every 10 seconds switches between red and green text. It also displays "Hello World" in the center of the screen.

How it works...

The `Main()` function in `AppMain.cs` is only changed slightly from the prior recipe. The major change is, instead of using a generic `Scene` object, we are now creating an instance of our custom `MyScene` object. The only other change is that we have centered the "Hello World" label in the middle of the screen.

`MyScene` is derived from `Sce.PlayStation.HighLevel.GameEngine2D.Scene`. We declare five member variables, one `Label` for each corner of the screen as well as a `float` for recording the time that elapsed since creation. In the constructor, we set the scene to render to the dimensions of the screen. We then cache the screen width and height for later use.

Next, we allocate all four of the labels, positioning each one slightly offset from each screen corner with the appropriate text. We then add each of the labels to `MyScene` object's widget collection using `AddChild()`. We then register `MyScene` to receive updates by passing our `MyScene` object to `Scheduler.Instance.ScheduleUpdateForTarget()`. We will cover this process in more detail in the next chapter.

The `Update()` method is called once per frame and simply adds the time delta (since the last call to `Update()`). In the `Draw()` method (which also is called once per frame), we first check if 10 seconds have elapsed. If it has, we set each corner label to green, then every 10 seconds we toggle between red and green.

There's more...

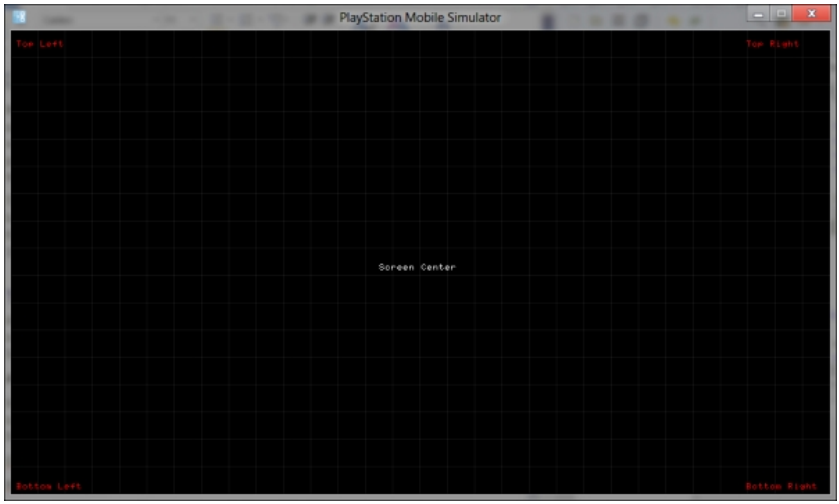
As you can see, GameEngine2D makes it easy to move logic from the game loop into individual scenes. By overriding the `Draw` method, you can control rendering at the scene level.

Adding a grid

It's quite common when debugging to display a grid over your scene. This task is incredibly easy to accomplish in a `Scene` object. At the bottom of your constructor add the following code:

```
Director.Instance.DebugFlags =
Director.Instance.DebugFlags |
DebugFlags.DrawGrid;
this.DrawGridStep = 32.0f;
```

This will cause a grid to appear when your scene is drawn, like this:



`DrawGridStep` determines the spacing of your grid in pixels.

You generally have to inherit from a `Node` derived object (like Scene) if you want to implement a custom `draw` function. You can, however, register a custom `Draw` method that will be called by `Director` each frame using the `AdHocDraw` event of any object that inherits from Node. For example:

```
_labelBottomLeft.AdHocDraw += () => {
 _labelBottomLeft.Color = new
Vector4(255.0f,255.0f,255.0f,255.0f);
};
```

This registers an `AdHocDraw` function to the node `_labelBottomLeft`, in this case a `lambda` function that changes the label's color to white. This function will be called every frame until the event handler is removed. This technique is used quite often in the PlayStation Mobile SDK samples to simplify the code. It prevents you from having to create a custom `Scene` if you simply want to override its `Draw` method.

See also

- See the *Handling updates with Scheduler* recipe in Chapter 4, *Performing Actions with GameEngine2D* for details on how to handle updating using `Scene` classes

Adding a sprite to a scene

In this recipe, we are going to create a sprite and add it to a scene, rendering it centered on the screen .

Getting ready

Create a new project and add a reference to GameEngine2D. You are going to need to add an image file to use as a sprite; I will be reusing our trusty `F18` graphic. The complete source for this example can be found in `Ch3_Example3`.

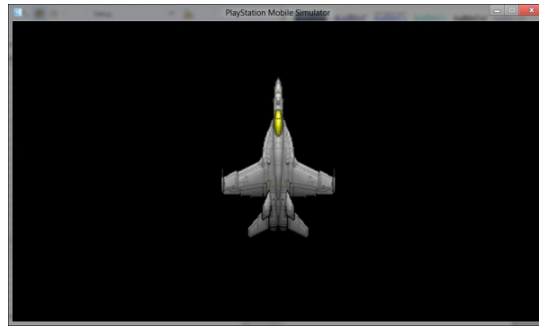
How to do it...

1. Open `AppMain.cs` and add the following code:

```
using System;
using
System.Collections.Generic;

using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Environment;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.Core.Input;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba


namespace Ch3_Example3
{
  public class AppMain
  {
    public static void Main
```

```
(string[] args)
    {
      Director.Initialize();

      Scene scene = new
Scene();

scene.Camera.SetViewFromViewport();

      TextureInfo ti = new
TextureInfo("/Application/
FA-18h.png");
      SpriteUV sprite = new
SpriteUV(ti);

      sprite.Scale =
ti.TextureSizef;
      sprite.Pivot = new
Vector2(0.5f,0.5f);
      sprite.Position = new
Vector2(

Director.Instance.GL.Context.Screen.Width

Director.Instance.GL.Context.Screen.Heigh


scene.AddChild(sprite);


Director.Instance.RunWithScene(scene);
    }
  }
}
```

2. Now run the code by hitting the *F5* key and you will see the following output:



How it works...

This code begins by initializing `Director`, creating, and setting up a scene as usual. We then create a `TextureInfo` object, passing it the path of our texture image. We then create a `SpriteUV` using the `TextureInfo` object.

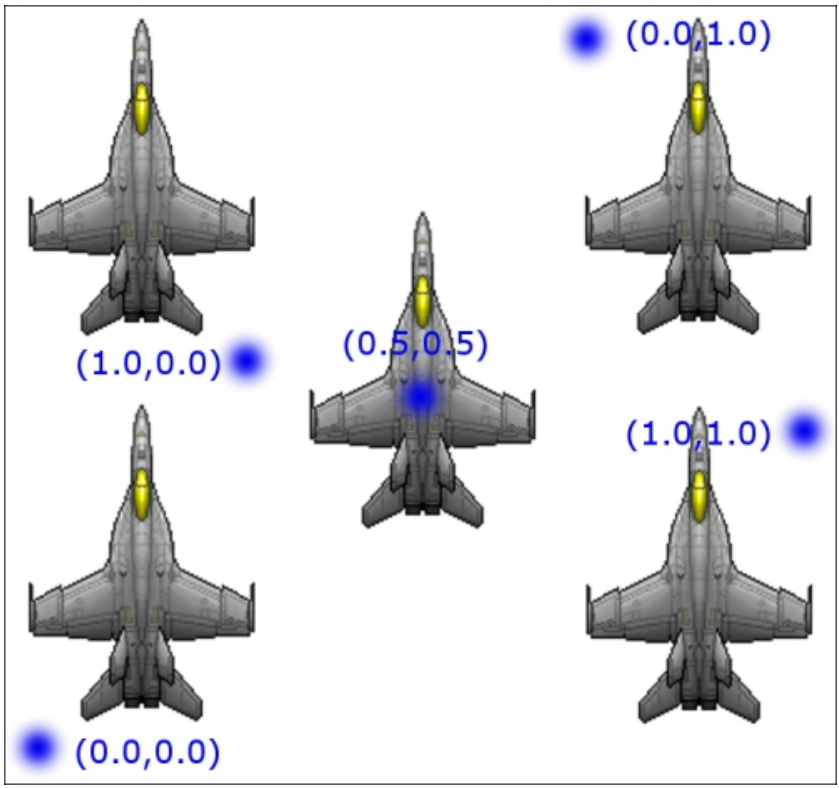
Next, we set our sprite's scale equal to its texture image size (in pixels). Then, we set its pivot point to the center of the sprite and position the sprite in the center of the screen. Finally, we add the sprite to our scene, then tell the `Director` method to run our scene, causing our sprite to be displayed.

There's more...

Pivot points are an important concept to understand. The pivot point is the position (within the sprite) that transformations are performed relative to. By default in GameEngine2D, the pivot point will be set to the bottom-left corner of the sprite. Therefore, if we translated (moved) the sprite to the top-right corner of the screen, it would be completely offscreen. If the pivot point was set to the middle of the sprite, the same transformation to the top-right corner of the screen would result in only the bottom-left corner of the sprite being visible, while the other three fourth of the sprite would be offscreen. When performing a rotation, the pivot point is the point the rotation is performed around.

The pivot point is a Vector2 value with valid values ranging from (0,0) to (1,1). A value of (0,0) being the bottom-left corner and the value (1,1) representing the top-right corner. The following diagram

demonstrates various pivot point values and their relative positions within the sprite:



Another important concept to understand is how transformations are actually applied to a `Sprite` object in GameEngine2D. Behind the scenes, we are still dealing with 3D and our sprite is actually a four-sided polygon aligned to the camera. This polygon data is held by the `Quad` attribute in `SpriteBase`, which `SpriteUV` inherits from. This `Quad` value in turn holds the four vertices that make up the quad, as well as the key attributes `T`, `R`, and `S`, which represent the current translation, rotation, and scale respectively. You do not generally access the `Quad` property directly; instead you use the higher-level properties such as Rotate and Scale inherited from the `Node` base class.

See also

- See the *Loading, displaying, and translating a textured image* recipe in Chapter 1, *Getting Started* for details on how to add an image resource to your project

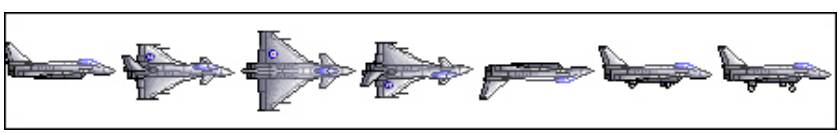
Creating a sprite sheet

In this recipe, we are going to create a sprite sheet using the freely available `TexturePacker` application.

Getting ready

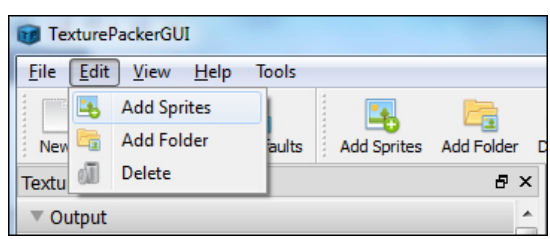
Head over to http://bit.ly/Nm0lUd and download `TexturePacker`. It is available for Window, Mac OS, and Ubuntu. This recipe uses the Windows version. The sprites used in this example can be downloaded from http://bit.ly/T3odB5. Run through the install and then run `TexturePackerGUI`. When prompted, select **Use Essential** to enable the basic free version.

We are going to create a 256 x 128 sprite sheet using these seven individual 64 x 64 pixel frames of animation, as shown in the following diagram:



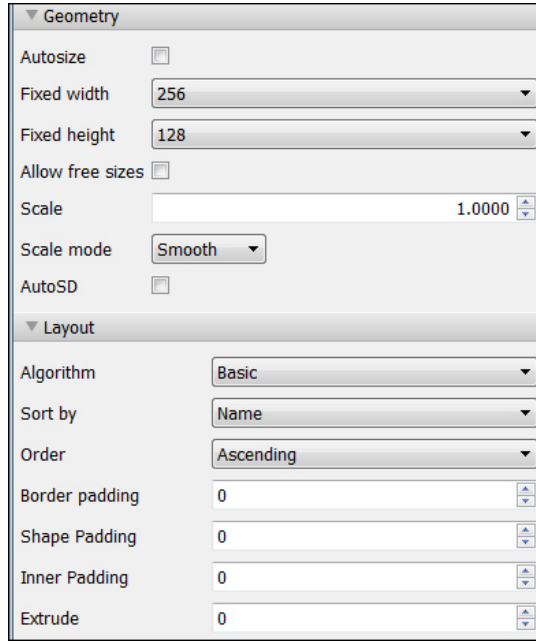
How to do it...

1. Select **Edit** | **Add Sprites**, as shown in the following screenshot:

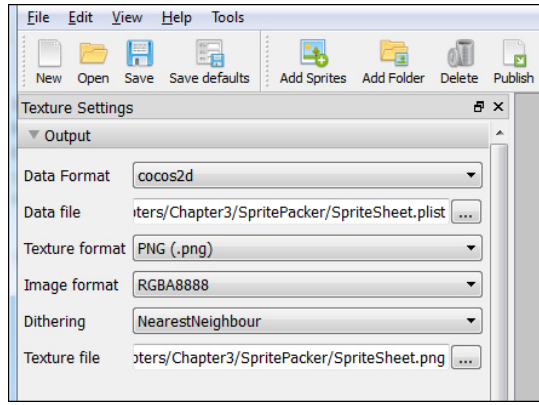


2. In the resulting dialog box, *Ctrl* + left-click to select our seven sprites in order and then click on **Open**.
3. By default, it will load the sprites to fit as tightly in the

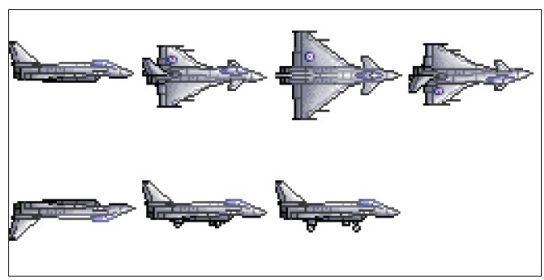
sprite sheet as possible. If we want to pack in a grid four sprites widely instead, in the **Texture Settings** panel under the **Layout** section, set **Algorithm** to **Basic**, **Border padding** to **0**, **Shape Padding** to **0**, and **Sort by** to **Name**. In the **Geometry** section set **Fixed width** to **256**, and **Fixed height** to **128**, as shown in the following screenshot:



4. In the **Output** section of the **Texture Settings** panel, specify a path and filename to save your sprite sheet to in the **Texture file** field, and then click on the **Publish** button at the top of the screen:



5. Your sprite sheet will be created in the location you specified in the previous step. It should now look something like the following screenshot:



How it works...

`TexturePacker` simply takes a sequence of independent image files and packs them together into a single image. When working with GPU hardware, power of 2 sizes (64 x 64, 128 x 128, 1024 x 1024, and so on) generally perform the best. In fact, some hardware actually requires textures to have width and height values that are a power of 2.

So why pack all of your textures together into a single texture? In a word, speed. Copying a single large file, either from the disk or in the memory, is almost always faster than copying a series of small files.

`TexturePacker` also generates a datafile describing the sprite

data and its positioning within the texture; the format this file is generated in is determined by the selection you choose under **Data Format**. In this case, we will not be using this file. If you want to pack as many files as possible within a texture sheet or want to store multiple different animation sequences in a single sheet, this datafile becomes critical for locating your sprites within the texture.

There's more...

`TexturePacker` is by no means the only option when it comes to generating sprite sheets. You can manually arrange your sprites into a single file using any image editor, such as Photoshop, Paint.NET, or GIMP. As well, there are a number of competing sprite creation tools, such as Sprite Sheet Packer (http://spritesheetpacker.codeplex.com/) and Zwoptex (http://bit.ly/NUtIA1).

See also

- See the *Loading, displaying, and translating a textured image* recipe in Chapter 1, *Getting Started* for information on the sprites used in this recipe
- See the *Using a sprite sheet in code* recipe for an example of how to make use of a sprite sheet with the PlayStation Mobile SDK

Using a sprite sheet in code

In this recipe we are going to look at playing an animation sequence using the sprite sheet we just created in the prior recipe.

Getting ready

This recipe assumes you have completed the *Creating a sprite sheet* recipe or have a sprite sheet acquired from another source. Make sure your sprite sheet is composed of sprites of an identical size. In this example, our sheet is 2 rows of 4 columns of sprites, with the bottom-right sprite being empty, for a total of seven sprites.

Create a new project, and then add a reference to GameEngine2D.

How to do it...

1. In `AppMain.cs` replace the existing code with the following code:

```
using System;
using Sce.PlayStation.Core;
using
Sce.PlayStation.HighLevel.GameEngine2D;
namespace Ch3_Example4
{
  public class AppMain
  {
    public static void Main
(string[] args)
    {
      Director.Initialize();
      SpritesheetScene scene
= new SpritesheetScene();

Director.Instance.RunWithScene(scene);
    }
  }
```

```
}
```

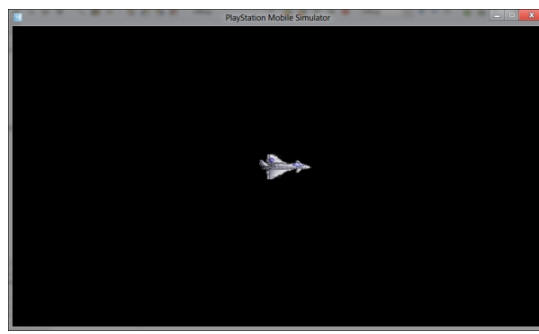
2. Now add a new file of type `Empty Class` named `SpritesheetScene.cs` to your project. Enter the following code:

```
using System;

using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba
namespace Ch3_Example4
{
  public class
SpritesheetScene : Scene
  {
    private SpriteTile
_currentSprite;
    private Texture2D
_texture;
    private TextureInfo _ti;

    private float
elapsedTime = 0.0f;

    public SpritesheetScene
()
    {
      _texture = new
Texture2D("/Application/
SpriteSheet.png",false);
      _ti = new
TextureInfo(_texture,new
Vector2i(4,2));
      _currentSprite = new
SpriteTile(_ti,new
```

```
Vector2i(0,1));
      _currentSprite.Pivot =
new Vector2(0.5f,0.5f);

_currentSprite.Position =
new Vector2(0.0f,0.0f);
      _currentSprite.Scale =
new Vector2(3.0f,3.0f);

this.AddChild(_currentSprite);


Scheduler.Instance.ScheduleUpdateForTarge
    }

    public override void
Update (float dt)
    {
      elapsedTime += dt;
      if(elapsedTime > 0.5f)
      {
        Vector2i currentTile
=
_currentSprite.TileIndex2D;
        if(currentTile.Y ==
1)
        {
          if(currentTile.X <
3)
            currentTile.X +
+;
          else
            currentTile =
new Vector2i(0,0);
        }
        else
        {
          if(currentTile.X <
2)
            currentTile.X++;
```

```
                else
                  currentTile =
new Vector2i(0,1);
            }

_currentSprite.TileIndex2D =
currentTile;
            elapsedTime = 0.0f;
          }
        }
      }
}
```

3. Hit *F5* to run your application and you should see the following output:



A jet sprite will display centered to the screen, then every half a second it will advance to the next frame until it runs out of animations, at which point it will start over again.

How it works...

The code in `AppMain.cs` should be familiar at this point. It simply initializes the `Director` singleton, creates an instance of our `SpritesheetScene` class, and then the `Director` method runs with the newly created scene.

`SpritesheetScene` is where the bulk of our activity occurs. We

start off by declaring four private member variables, a `SpriteTile` object to represent the currently displayed sprite within the sprite sheet, a `Texture2D` object to store the actual image in memory, a `TextureInfo` object to store texture details such as UV coordinates and number of sprites, and finally a `float` variable to keep a running total of time that has elapsed.

In our constructor we load `_texture` with a call to `Texture2D` specifying our texture's file location and the fact we do not want a mipmap generated. We then create our `TextureInfo` object, `_ti` by passing in our newly generated texture as well as a `Vector2i` specifying the number of sprites comprising our sprite sheet. It is very important to note that this value is *NOT* zero based.

We then create a `SpriteTile` object, `_currentSprite` passing in `_ti`, as well as specifying the indices of the sprite within the sprite sheet we want the `SpriteTile` object to point at. In this case, this value *is* zero based. When we specify (0,1) we are pointing at the top-left sprite in our sprite sheet, keeping in mind that GameEngine2D coordinates start at the bottom-left corner while our image file starts at the top-left. Next, we set the pivot point of the sprite to its center, its position to the middle of the screen, and then scale it up three times in size just to make it a bit more visible. Finally, we add our sprite to the scene and then schedule our scene to receive updates. We will cover the updating process in more detail in the next chapter.

In the `update` method, we increment the `_elapsedTime` variable by value of `dt`, which is the amount of time that has occurred since the last time update was called. We then check if the elapsed time is greater than *.5* (half a second). If half a second has elapsed, we then advance the sprite one tile to the right in the animation sequence. If advancing would cause us to exceed the sprites available, we instead roll over to the beginning of the next now. Keep in mind that on the bottom row (`currentTile.Y == 0`) we check against `currentTile.X`, if it is less than *2* instead of *3*, because the bottom-right sprite in our sprite sheet is empty. Once we have performed the calculation, we update the sprite that `_currentSprite` is pointing at by updating `TileIndex2D`. We then reset `_elapsedTime` to zero to begin the half-second

countdown all over again.

There's more...

This whole process probably looked a bit more complicated than it actually is. Essentially you create a sprite sheet by passing in a `Vector2i` variable to your `TextureInfo` object, letting it know the texture it represents is actually a sprite sheet, containing however many rows and columns you specified in the vector. A `SpriteTile` object points at a single sprite within the sheet. When you create the `SpriteTile` object you can specify a `Vector2i` variable, which sets the offset from which the `SpriteTile` object will get its sprite. This value can later be updated using the value `TileIndex2D`. Think of the sprite sheet as a grid of sprites, and `TileIndex2D` as coordinates for an individual cell within that grid.

It is important to understand how tile indexing works. A sprite sheet requires all the images to be of the same width and height for a reason. When you tell a `TextureInfo` object that it holds a sprite sheet, it creates texture coordinates based on the dimensions you pass in. Say for example you pass in (5,5) to a `TextureInfo` object that contains a 160 x 160 texture; it will effectively hold 25 sprites with a 32 x 32 texture, in 5 rows, 5 columns wide. Using `TileIndex2D` to access the sprites, the values (0,0) to (4,0) will represent the bottom row of sprites in the image, (0,1) to (4,1) will represent the next row up, while (4,4) would represent the top-right 32 x 32 pixel tile within the sprite sheet. You can also access the individual sprites using `TileIndex1D`. In this case *0* represents the bottom-left sprite within the sprite sheet, *5* would represent the sprite directly above it, while *24* would represent the top-right sprite.

One of the major advantages to using a sprite sheet is the `SpriteTile` class. This class is a lot *lighter* than the `SpriteUV` class, as all of the sprites within a `SpriteTile` class have the same UV data. This means it will load quicker and use less memory and CPU than a comparable `SpriteUV` class.

See also

- See the *Creating a sprite sheet* recipe for instructions on

how to create a sprite sheet

Batching a sprite with SpriteLists

In this recipe we are going to look at the benefits of using `SpriteLists` to accelerate 2D rendering of similar objects.

Getting ready

Create a new project and add a reference to GameEngine2D. Add a sprite to your project. I am going to reuse our trusty `FA-18H` sprite from prior recipes. You can download the complete source at `Ch3_Example5`.

How to do it...

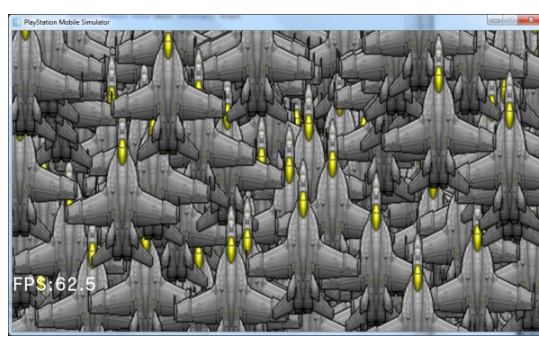
1. In `AppMain.cs` enter the following code:

```
using System;
using
System.Collections.Generic;

using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba
namespace Ch3_Example5
{
  public class AppMain
  {
    public static void Main
(string[] args)
    {
      Director.Initialize();
      Scene scene = new
```

```
Scene();

scene.Camera.SetViewFromViewport();
      Texture2D texture =
new Texture2D("/Application/
FA-18H.png",false);
      TextureInfo ti = new
TextureInfo(texture);
      List<SpriteUV> sprites
= new List<SpriteUV>();

      Random random = new
Random();

      SpriteList spriteList
= new SpriteList(ti);
      for(int i = 0;i<
1000;i++)
      {
        SpriteUV sprite =
new SpriteUV(ti);
        sprite.Pivot = new
Vector2(0.5f,0.5f);
        sprite.Position =
new Vector2(

random.Next(0,Director.Instance.GL.Contex

random.Next(0,Director.Instance.GL.Contex
        sprite.Scale =
sprite.TextureInfo.TileSizeInPixelsf;

        sprite.Schedule(
(dt) => {

if(sprite.Position.Y <
Director.Instance.GL.Context.Screen.Heigh
          {
            sprite.Position
= new
```

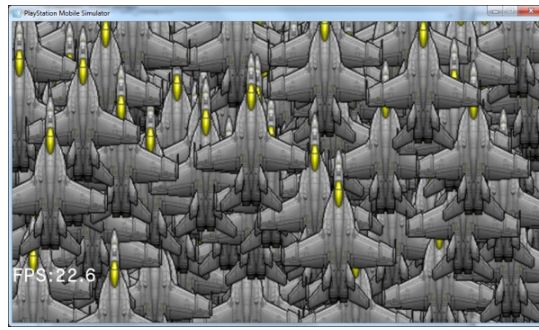
```
Vector2(sprite.Position.X,sprite.Positior
+ 1);
                }
                else
                {
                  sprite.Position
= new
Vector2(sprite.Position.X,0.0f);
                }
              });

spriteList.AddChild(sprite);
            }

            scene.AddChild
(spriteList);
            scene.AddChild(new
FPS());

Director.Instance.RunWithScene(scene);
          }
        }
}
```

2. Run the code by hitting *F5* and you will see the following output:



3. Now uncomment the line `spriteList.AddChild(sprite);` and remove the comment before the line `//`

`scene.AddChild(sprite);`. Run it again to see the following output:



Running in the simulator on my laptop, the sprite list version runs at least three times faster. This is especially impressive as the simulator has a fixed frame rate cap of *60 fps*.

How it works...

We start off with the standard initialization code, setting up `Director`, our scene, and creating our `Texture` and `TextureInfo` objects. We then declare a `SpriteList` object. Next we create a list of `SpriteUV` objects, then loop 1,000 times creating a new sprite, setting it up, and positioning it randomly within the screen. We then schedule a `lambda` function for each sprite that will be called during the update phase, something we will cover in more detail in the next chapter. Finally, we either add the newly created sprite to the `SpriteList` (or to the scene normally if we uncomment the line `scene.AddChild(sprite);`).

Now that we have created 1,000 sprites and added them to `SpriteList`, we add `SpriteList` to our scene. We also add a new **FPS** (**Frames Per Second**) widget to the scene to display the current frame rate. Finally, we tell the Director singleton to run using `scene`.

The simple act of adding our sprites to the `SpriteList`, which is then added to the scene, instead of adding the sprites to the scene directly, resulted in a massive increase in speed.

There's more...

`SpriteList` manages this massive increase in speed by grouping all of the sprites together in a single batch. If you are coming from an XNA background, `SpriteList` is PlayStation Mobile's equivalent of the `SpriteBatch` class, and it has similar restrictions.

`SpriteList` gains this massive increase in speed by drawing all of the sprites using the exact same settings. This removes a lot of rendering overhead and setup from the process but imposes some serious restrictions. First, you can only add `SpriteUV` or `SpriteTile` derived objects to the `SpriteList` class. Second, all of the objects in a `SpriteList` class need to have the same `TextureInfo` object. The most common real world scenario for using `SpriteList` is to organize your game sprites into a number of different sprite sheets, then having one `SpriteList` per sprite sheet. You can have as many `SpriteList` objects in your scene as you want; however, you will benefit less and less the fewer the number of sprites in a SpriteList. If your sprite is only going to have a few instances on screen at once, do not use a `SpriteList` object.

We also made use of a simple widget I wrote called FPS. This is a simple `SpriteUV` derived object that creates an image with the current frame rate. Here is the code for `FPS.cs`:

```
using System;

using Sce.PlayStation.Core;
using Sce.PlayStation.Core.Graphics;
using Sce.PlayStation.Core.Imaging;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Base;

namespace Ch3_Example5
{
 public class FPS : SpriteUV
 {
```

```
  TextureInfo _ti;

  public FPS ()
  {
   Texture2D texture = new
Texture2D(150,100,false,

PixelFormat.Rgba);
   _ti = new TextureInfo(texture);

   this.TextureInfo = _ti;
   this.Quad.S = new
Sce.PlayStation.Core.Vector2(150,100);

Scheduler.Instance.ScheduleUpdateForTarget(this,1
  }

  public override void Update (float
dt)
  {
   _ti.Dispose();
   Image img = new
Image(ImageMode.Rgba, new
ImageSize(150,100),
                          new
ImageColor(255,255,255,0));
   img.DrawText("FPS:" + (1/
dt).ToString(),
                new
ImageColor(255,255,255,255),
                new
Font(FontAlias.System,32,FontStyle.Bold),
                new
ImagePosition(0,0));

   Texture2D texture = new
Texture2D(150,100,false,

PixelFormat.Rgba);
```

```
texture.SetPixels(0,img.ToBuffer(),PixelFormat.Rg
    img.Dispose();
    _ti = new TextureInfo(texture);
    this.TextureInfo  = _ti;

    base.Update (dt);
   }
   ~FPS(){
    _ti.Dispose();
   }
  }
}
```

Simply add `FPS.cs` to your project, create a new FPS, and add it to your scene, and it will automatically update with the current frame rate.

For performance reasons, it is incredibly important to batch together calls with similar properties (same shader, render settings, texture, and so on), not just on PSM, but in game programming in general. Removing the setup overhead between common rendering calls can make a huge difference, the difference between a good frame rate and a game that struggles when drawing to the screen.

See also

- See the *Using a sprite sheet in code* recipe for more details on using `SpriteTile` and sprite sheets

Manipulating a texture's pixels

In this recipe, we are going to modify a sprite by working with its pixel data directly. We are going to perform two actions on the image. First we are going to increase the transparency of every solid pixel in the image. Then we are going to surround the image with a red rectangle, as you might put around a selected character sprite or button.

Getting ready

Create a new solution, add a reference to GameEngine2D, and add a sprite object. Once again I am going to use `FA-18H.png`. The source for this project is available as `Ch03_Example6`.

How to do it...

1. In `AppMain.cs` replace the existing code with the following code:

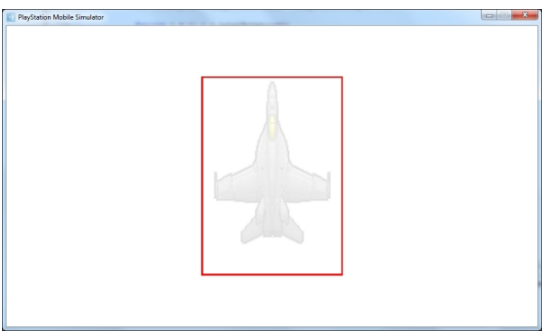
```
using System;
using
System.Collections.Generic;

using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.Core.Imaging;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba

namespace Ch03_Example5
{
  public class AppMain
  {
```

```
    public static void Main
(string[] args)
    {
      Director.Initialize();
      Scene scene = new
Scene();

scene.Camera.SetViewFromViewport();

      Image image = new
Image("Application/
FA-18H.png");
      image.Decode();
      var buffer =
image.ToBuffer();

      int totalBytes =
image.Size.Width *
image.Size.Height * 4;

      for(int i = 0; i <
totalBytes;i+=4)
        if(buffer[i+3] !=
(byte)0)
          buffer[i+3] =
(byte)32;

      for(int i = 0; i <
totalBytes; i+=4)
      {
        if(i <
image.Size.Width * 4 * 3 ||
           i >= (totalBytes
- (image.Size.Width * 4 *
3)) ||
           (i %
(image.Size.Width * 4)) < 12
||
           (i %
(image.Size.Width * 4)) >=
```

```
(image.Size.Width * 4) - 12
            )
          {
            buffer[i] =
(byte)255;
            buffer[i+3] =
(byte)255;
          }
        }


      Texture2D texture =
new
Texture2D(image.Size.Width,


image.Size.Height,

            false,


PixelFormat.Rgba);

texture.SetPixels(0,buffer);
      TextureInfo ti = new
TextureInfo(texture);

      SpriteUV sprite = new
SpriteUV(ti);
      sprite.Scale =
ti.TextureSizef;
      sprite.Pivot = new
Vector2(0.5f,0.5f);

      sprite.Position = new
Vector2(

Director.Instance.GL.Context.Screen.Width

Director.Instance.GL.Context.Screen.Heigh
```

```
            scene.AddChild(sprite);


            Director.Instance.GL.Context.SetClearColo
            Vector4(255,255,255,255));

            Director.Instance.RunWithScene(scene);
                }
              }
            }
```

2. Run your scene by hitting *F5* and you will see the following output:



How it works...

After the standard `Director` and `Scene` initialization, instead of loading an image directly into a `Texture2D` object, we instead create an `Image` object, located in the `Sce.PlayStation.Core.Imaging` namespace. We create an image by passing it a file path as usual, then decode the image from its native (often compressed) format using a call to `image.Decode();`.

At this point we get the image data by calling `ToBuffer()`, which returns a byte array we store as buffer. This buffer is going to be composed of 4 byte pairs representing each pixel, one byte for each color (red, green, blue), plus one byte for the alpha channel (transparency).

The first four bytes in the buffer are the top-left pixel in the image, while the last four bytes represent the bottom-right corner pixel of the image. We can therefore calculate the total size of our byte buffer with the following code:

```
int totalBytes = image.Size.Width *
image.Size.Height * 4;
```

First, we loop through all of the pixels in the image and if the alpha channel isn't 0 (meaning it isn't completely transparent already), we reduce (or increase) its alpha channel to 12.5 percent (32/256), making it almost completely transparent. Remember, each pixel is represented by four bytes, so `buffer[0]` is the red pixel, `buffer[1]` is the green pixel, `buffer[2]` is the blue pixel, and `buffer[3]` is the alpha channel. This is why we increment the index by four at a time each iteration through the loop, causing `i` to point to the next pixel grouping in the buffer.

Next, we want to draw a red *3* pixel border around the outside edge of the sprite. We do this by checking our location in the array. We can tell our position by calculating our offset within the byte array. Each "line" of image data will be *image width X 4 bytes per pixel bytes* long, so a *100 x 100* image would be composed of *100 x 400* byte lines. If we are in the first three lines, the last three lines, or within *12* bytes (*3* pixels) of either the left or right side of the image, we set the red byte to full (*256*). The end result is a red rectangle *3* pixels thick around the edge of our image.

Now, that we have modified our image buffer, we create a `Texture2D` object using the image dimensions, then copy in the pixel data using `SetPixels()`. The first parameter represents the image layer and will always be 0, unless you are using mipmapping. We then create `TextureInfo` and `SpriteUV` objects, center them to the screen, tell the `Director` object to clear to white each frame it draws, then finally start our scene running.

There's more...

It is important to note that this entire recipe assumes you are using a

RGBA pixel format image, which is the default. If you use a different pixel format, all of the calculations need to be updated! The PlayStation Mobile SDK supports a number of different pixel formats, but RGBA is the norm, as it is the default in most graphic applications.

Many of you will be tempted to work directly with pixels; there is something primal about working directly with your image data! I highly recommend against this! Modern CPUs do not really work at the pixel level anymore, so direct pixel level manipulations are a rather expensive action. Try to keep any direct pixel operations done on a frame by frame basis to an absolute minimum.

See also

- See the *"Hello World" drawing text on an image* and *Manipulating an image dynamically* recipes in Chapter 1, *Getting Started* for more details on using the imaging classes

Creating a 2D particle system

In this recipe, we are going to create an extremely simple pinwheel particle system.

Getting ready

Create a new solution, and add a reference to GameEngine2D. You will also need a small graphic to use as your particle; I created a small 32 x 32 pixel image that was mostly transparent. Create an image and add it to your project. The source for this project is available as `Ch03_Example7`.

How to do it...

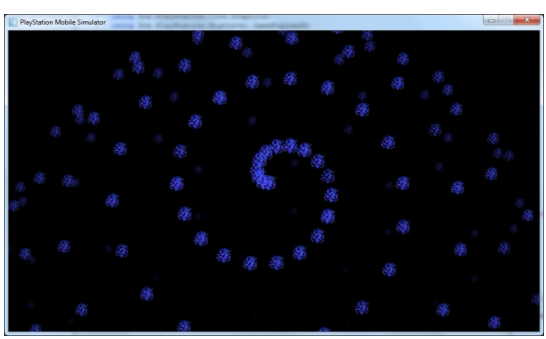
1. In `AppMain.cs` replace the existing code with the following code:

```
using System;
using
System.Collections.Generic;

using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba

namespace Ch3_Example7
{
  public class AppMain
  {
    public static void Main
(string[] args)
    {
      Director.Initialize();
```

```
      Scene scene = new
Scene();

scene.Camera.SetViewFromViewport();

      var particles = new
Particles(2000);
      var pinwheel =
particles.ParticleSystem;
      pinwheel.TextureInfo =
new TextureInfo(new
Texture2D("Application/
WaterDrop.png",false));

      pinwheel.Emit.Position
= new Vector2(

Director.Instance.GL.Context.Screen.Width

Director.Instance.GL.Context.Screen.Heigh
      pinwheel.Emit.LifeSpan
= 5.0f;
      pinwheel.Emit.Velocity
= new Vector2(0.0f,650.0f);
      pinwheel.Emit.WaitTime
= 0.005f;

pinwheel.Emit.ScaleStart =
26.0f;

pinwheel.Emit.ScaleEnd=
6.0f;
      pinwheel.Emit.ColorEnd
= new
Vector4(0.0f,0.0f,0.0f,0.0f);


pinwheel.Simulation.Gravity
= 300.0f;
```

```
pinwheel.Simulation.Fade =
0.0f;
      float rotate = 0.0f;

particles.Schedule((dt) => {

pinwheel.Emit.Velocity =
pinwheel.Emit.Velocity.Rotate
(

Sce.PlayStation.HighLevel.GameEngine2D.Ba
+=1.0f*dt));
      });



scene.AddChild(particles);

Director.Instance.RunWithScene(scene);
    }
  }
}
```

2. Run your scene with *F5* and you will see the following output:



How it works...

We start off with the standard `Director` and `Scene` initialization code. We then create a `Particles` object, passing it 2000, which

represents the maximum particle count that the system will support. The `Particles` class is just a simple wrapper around a `ParticleSystem` object, enabling the `Particles` object to be used as a node within a scene. The actual heavy lifting is done in the `ParticleSystem` attribute, so we take a reference for convenience in the variable `pinwheel`.

First, we create a `TextureInfo` object that will be used for each particle. The `TextureInfo` object contains a `Texture2D` object, which in turn loads the image we created, in this case `WaterDrop.png`. This image is going to be the visual basis of every rendered particle.

Now that we have an image for our particle system to draw each particle with, we configure the particle emitter. This is done using the `Emit` attribute. We set the `LifeSpan` of each particle to `5`, causing the particles to live for 5 seconds. We then set the `Velocity` to (0.0f, 650.0f), which causes the particles to shoot straight up and out to a distance of 650 pixels (minus the effect of gravity). Next, we set the `WaitTime` to `0.005`, which is the time in seconds to wait before creating each particle, resulting in a nearly constant stream of particles. Then, we set the `ScaleStart` and `ScaleEnd` values. These values represent the size of the particles when they are emitted; then the values will scale down, resulting in each particle shrinking over its 5 second lifespan. Finally, we set the `ColorEnd` variable to a transparent black, causing particles to fade to black at the end of their lifespans.

Next, we set up the simulation parts of the particle system. In this case we tell it to apply `Gravity` at a rate of *300* pixels per second. The axis that gravity is applied on can also be specified (using `GravityDefault`), but in this case we will take the default. Finally, we tell the particles not to `Fade`, as we do this already (and smoother) using the `ColorEnd` attribute of the `Emit` class. We then schedule a simple update lambda to be called each frame during the scheduler's update phase (we will cover this in more detail in the next chapter). Each update we apply a crude rotation to the emitter, causing the pinwheel effect as the emitter rotates each frame. Finally, we add the `Particles` object to our scene and start it running.

There's more...

Due to space constraints, this recipe only brushes on the very basics of what `ParticleSystem` is capable of. There are a few dozen other settings we haven't mentioned here. In addition, you can apply a shader to each particle, resulting in just about any effect you can imagine. Particles can be used to represent explosions, fire, water effects, and so on.

See also

- The PlayStation Mobile SDK example `Feature Catalog` in the `\sample\GameEngine2D` folder demonstrates a fire/smoke effect using a compound particle system

Chapter 4. Performing Actions with GameEngine2D

In this chapter we will cover:

- Handling updates with Scheduler
- Working with the ActionManager object
- Using predefined actions
- Transitioning between scenes
- Simple collision detection
- Playing background music
- Playing sound effects

Introduction

In this chapter, we are going to explore the various ways you can control program execution using GameEngine2D. This involves the use of two critical subsystems, **ActionManager** and **Scheduler**. We are going to look at how you can make use of both these systems. We are also going to look at how you play sound effects and background music, as well as see how to add a graphical transition between scenes.

Handling updates with Scheduler

In this recipe we are going to demonstrate three different ways you can update a node derived object: handling the update directly in the game loop, inheriting from a node derived class, and overriding the `Update` method or passing a delegate in to a `Node's Schedule` property.

Getting ready

Load up PSM Studio and create a new project. Add a reference to `Sce.PlayStation.GameEngine2D`. We also need a sample sprite to work with; I will again be using our trusty `F18` sprite, but feel free to substitute any reasonably sized graphic of your own. The complete source for this example can be found in `Ch4_Example1`.

How to do it...

1. Add the sprite to the project and set its **Build Action** to **Content**.
2. Open `AppMain.cs` and replace the existing code with the following code:

```
using System;
using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba

namespace Ch4_Example1
{
  public class AppMain{
    public static void Main
```

```
(string[] args){
      Director.Initialize();
      Scene scene = new
Scene();

scene.Camera.SetViewFromViewport();

      SpriteUV jetSprite1 =
new SpriteUV(new
TextureInfo("/Application/
FA-18h.png"));
      jetSprite1.Scale =
jetSprite1.TextureInfo.TextureSizef.Multi
Vector2(0.5f,0.5f));
      jetSprite1.Position =
new Vector2(0,550 -
jetSprite1.Scale.Y);

      SpriteUV jetSprite2 =
new JetSpriteWithUpdate();
      SpriteUV jetSprite3 =
new SpriteUV(new
TextureInfo("/Application/
FA-18h.png"));

      jetSprite3.Scale =
jetSprite3.TextureInfo.TextureSizef.Multi
Vector2(0.5f,0.5f));
      jetSprite3.Position =
new Vector2(0,150 -
jetSprite3.Scale.Y);


jetSprite3.Schedule((dt) =>
{

if(jetSprite3.Position.X >
Director.Instance.GL.Context.GetViewport(

jetSprite3.Position = new
```

```
Vector2(0,150 -
jetSprite3.Scale.Y);
        else

jetSprite3.Position = new
Vector2(jetSprite3.Position.X
+ 1,150 -
jetSprite3.Scale.Y);
      });

scene.AddChild(jetSprite1);

scene.AddChild(jetSprite2);

scene.AddChild(jetSprite3);

Director.Instance.RunWithScene(scene,true

      bool done=false;
      while(!done){

Director.Instance.Update();


if(jetSprite1.Position.X >
Director.Instance.GL.Context.GetViewport(

jetSprite1.Position = new
Vector2(0,550 -
jetSprite1.Scale.Y);
        else

jetSprite1.Position = new
Vector2(jetSprite1.Position.X
+ 1,550 -
jetSprite1.Scale.Y);


Director.Instance.Render();
```

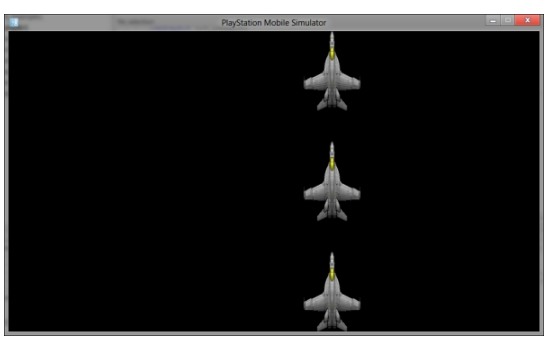
```
            Director.Instance.GL.Context.SwapBuffers(

            Director.Instance.PostSwap();
                }
            }
        }
    }
```

3. Create a new `CS` file named `JetSpriteWithUpdate.cs` and enter the following code:

```
using System;
using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba

namespace Ch4_Example1
{
  public class
JetSpriteWithUpdate :
SpriteUV
  {
    public
JetSpriteWithUpdate ()
    {
      this.TextureInfo = new
TextureInfo("/Application/
FA-18H.png");
      this.Scale =
this.TextureInfo.TextureSizef.Multiply(ne
Vector2(0.5f,0.5f));
      this.Position = new
Vector2(0,350 -
this.Scale.Y);
```

```
this.ScheduleUpdate(0);
    }

    public override void
Update (float dt)
    {
      if(this.Position.X >
Director.Instance.GL.Context.GetViewport(
        this.Position = new
Vector2(0,350 -
this.Scale.Y);
      else
        this.Position = new
Vector2(this.Position.X +
1,350 - this.Scale.Y);
    }

    ~JetSpriteWithUpdate()
    {

this.TextureInfo.Dispose();
    }
  }
}
```

4. Press *F5* to run and you will see the following output:



What you should see is three individual jet sprites, all being updated identically each frame by a completely different method.

How it works...

This code starts off with typical boilerplate code. We initialize our `Director` object, create a new scene, and then set the size of the camera to match the viewport. Next, we create three different instances of the same `F18` sprite, each positioned evenly across the left-hand side of the screen. The first sprite, `jetSprite1`, we are going to update the traditional way, in the game loop. The second sprite is a custom class derived from `SpriteUV`, which we will look at in a second.

The final sprite is controlled using its `Schedule` member. In this case, we are simply passing a lambda function that is going to move the sprite left to right, until it moves beyond the edge of the screen, then it is going to repeat the process all over again. The function takes a single parameter, `dt`, which is a `float` variable containing the elapsed time since the last call in seconds.

Now take a look at the `JetSpriteWithUpdate` class. This class inherits from `SpriteUV`; in the constructor we simply perform all of the tasks we did on the standalone sprites, load the sprite, scale, and position it. Finally, we call `ScheduleUpdate()`, which registers the sprite's `Update` method with the global `Scheduler` object. The logic in the `Update` method is identical to that we used for the other sprites. Finally, we declare a destructor, because `TextureInfo` is an `IDisposable` resource and needs to be disposed off manually.

Back in `AppMain.cs`, we add all three sprites to our scene, then kick the scene off with a call to `RunWithScene()`. We pass in the value `true` because we want to manage our game loop manually.

As you may recall from the previous chapter, if you manually handle a game loop, there are four `Director` methods that need to be called. We perform an infinite `while` loop, calling `Update()`, `Render()`, `SwapBuffers()`, and `PostSwap()`. In between `Update()` and `Render()`, we manually update `jetSprite1`, using the same logic again.

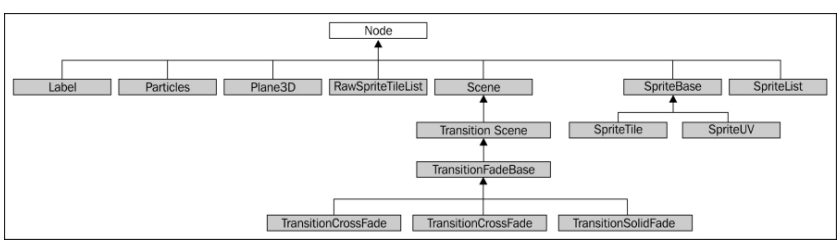
There's more...

The call by `Director` to `Update()` is very important, as behind the scenes this is what tells the `Scheduler` object to update, which in turn causes all of the objects and functions registered with `Scheduler` to get updated. If you remove this call from the game loop, no updating will occur.

In addition to `Schedule()`, `Node` has a method named `ScheduleInterval()`, which takes an additional parameter interval, which tells the scheduler how often (in seconds) to call the scheduled function. There is also a method named `ScheduleUpdate()`, which performs the exact same process as `Schedule`, but instead passes the Node's `Update` method instead of a user defined function, as we saw in `JetSpriteWithUpdate`.

`Scheduler` itself is a singleton object that we have not used directly in this recipe. `Scheduler` can be accessed the same way as the `Director` singleton using the `Instance` property, although `Scheduler` does not require initialization. It exposes members for adding and removing scheduled updates, although you have very little reason to utilize it directly.

As mentioned earlier, you can perform updates on any `Node` derived object. The following is the class hierarchy of classes derived from node. You can schedule updates on any of these classes:



See also

- See the *A game loop, GameEngine2D style* recipe in Chapter 3, *Graphics with GameEngine2D* for more details on the game loop process

Working with the ActionManager object

In this recipe, we are going to look at using the `ActionManager` singleton to perform actions on nodes. We will create our own action, `OrbitTarget`, which enables our moon sprite to orbit our earth sprite.

Getting ready

Load up PlayStation Mobile Studio and create a new project. Add a reference to `Sce.PlayStation.GameEngine2D`. We also need to add a pair of sprites to the project, one representing the moon, the other the earth. The complete source and images for this example can be found in `Ch4_Example2`.

How to do it...

1. Open `AppMain.cs` and replace the existing code with the following code:

```
using System;
using
System.Collections.Generic;

using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Environment;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.Core.Input;

using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba
```

```
namespace Ch4_Example2
{
  public class AppMain{
    public static void Main
(string[] args){
      Director.Initialize();
      Scene scene = new
Scene();

scene.Camera.SetViewFromViewport();
      var screenSize =
Director.Instance.GL.Context.GetViewport(
      SpriteUV earth = new
SpriteUV(new TextureInfo("/
Application/earth.png"));
      earth.Scale =
earth.TextureInfo.TextureSizef;
      earth.Pivot = new
Vector2(0.5f,0.5f);
      earth.Position = new
Vector2(
        screenSize.Width/2,

screenSize.Height/2);

      SpriteUV moon = new
SpriteUV(new TextureInfo("/
Application/moon.png"));
      moon.Scale =
moon.TextureInfo.TextureSizef;
      moon.Pivot = new
Vector2(0.5f,0.5f);
      moon.Position = new
Vector2(
        screenSize.Width/2,
        screenSize.Height -
moon.Scale.Y/2);

      scene.AddChild(earth);
```

```
            scene.AddChild(moon);

            OrbitTarget orbit =
new OrbitTarget(earth,
180.0f);

ActionManager.Instance.AddAction(orbit,mo
            orbit.Run();

Director.Instance.RunWithScene(scene);
        }
    }
}
```

2. Create a new file, `OrbitTarget.cs`, and enter this code:

```
using System;
using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba
namespace Ch4_Example2
{
  public class OrbitTarget :
ActionBase
  {
    Node _nodeToOrbit =
null;
    float _currentAngle =
0.0f;
    float _degreesPerSecond;


    public OrbitTarget (Node
nodeToOrbit, float
degreesPerSeconds)
```

```
    {
      _nodeToOrbit =
nodeToOrbit;
      _degreesPerSecond =
degreesPerSeconds;
    }

    public override void
Update (float dt)
    {
      _currentAngle+= dt*
Sce.PlayStation.HighLevel.GameEngine2D.Ba

      var distance =
this.Target.Position.Distance(this._nodeT

      var newAngleX =
(float)System.Math.Cos(_currentAngle);
      var newAngleY =
(float)System.Math.Sin(_currentAngle);

      var targetPos = new
Vector2(

_nodeToOrbit.Position.X +
(distance  * newAngleX),

_nodeToOrbit.Position.Y +
(distance  * newAngleY)

        );
      this.Target.Position =
targetPos;
    }
  }
}
```

3. Now hit *F5* to run your application and you should see the following output:

The moon will rotate around the earth at a fixed orbit of half a complete rotation per second.

How it works...

First, we initialize our `Director` object, create a new scene and size its camera to match the viewport dimensions. We then create a `SpriteUV` object for our earth and moon sprites and set each one's scale, pivot, and position values. We want the earth sprite positioned in the middle of the screen, while the moon sprite is going to start at the top center of the screen. We then add both sprites to our scene.

Next, we create an instance of our `OrbitTarget` class, passing in the object we want to orbit (the earth sprite), as well as the amount in degrees that we want to orbit by each second. We then register our action with a call to the `ActionManager` singleton's `AddAction()` method, passing both the action we want to perform as well as the node to perform the action on. We then tell our action to run and finally run the scene.

In `OrbitTarget.cs` we declare a new class `OrbitTarget`, derived from `ActionBase`. It has three member variables: `_nodeToOrbit` is the `Node` object passed in via the constructor and is the object that we want to orbit around, `_currentAngle` stores the current rotation (relative to the orbited node), and `_degreesPerSecond` is the speed to orbit, also passed in to the constructor. In the constructor, we simply store the target node and degrees per second value.

The bulk of the logic is in the `Update()` method, which is going to

be called each frame (after the action has been `Run()`). It is passed to the float `dt`, which is the elapsed time in seconds since the last time `Update()` was called.

In `Update` we increment the `_currentAngle` value by the degrees per second converted to radians, multiplied by the elapsed time. By calculating by the elapsed time, you are essentially splitting your movement up over fractions of a second, so that after a second elapses you should have moved by the desired amount.

The formula for rotating one object around another object is *X = distance + Cos(angle)*, *Y = distance + Sin(angle)*, plus the coordinates of the object to orbit. This code performs that calculation. We then update our position to the newly rotated value.

There's more...

An action's target property is the node the action is applied to. This value is passed as the second parameter when calling `ActionManager.Instance.AddAction()`.

`ActionManager`, such as the `Scheduler`, is called each frame during the `Director.Update()` phase of the game loop. You should use `Scheduler` to schedule objects to receive updates, while you should use `ActionManager` to actually perform the actions. In real production code, the previous recipe using the `Scheduler` object would actually be better performed using actions.

As you can see from the recipe, if you do not manually manage the game loop, the default game loop will automatically call the `Director` object's `Update()` method. You can therefore move all game logic into individual scene `Update` methods, and handle moving even more game logic into reusable activity classes. Actions make it extremely easy to write reusable game logic. Actions can only be applied to classes derived from the `Node` class (which includes `Scene` and `SpriteUV`).

In addition to `ActionBase`, custom actions can inherit from the following classes:

- `ActionWithDuration`

- `ActionTweenGeneric<T>`
- `ActionTweenGenericVector2`
- `ActionTweenGenericVector2Rotation`
- `ActionTweenGenericVector2Scale`
- `ActionTweenGenericVector4`

See also

- See the *Using predefined actions* recipe for examples of using GameEngine2D's built in action classes

Using predefined actions

In this recipe, we are going to demonstrate the use of a number of the built-in actions available in GameEngine2D.

Getting ready

Load up PlayStation Mobile Studio and create a new project. Add a reference to `Sce.PlayStation.GameEngine2D`. We are going to re-use the earth sprite from the previous recipe, but you can substitute any other image. The complete source and images for this example can be found in `Ch4_Example3`.

How to do it...

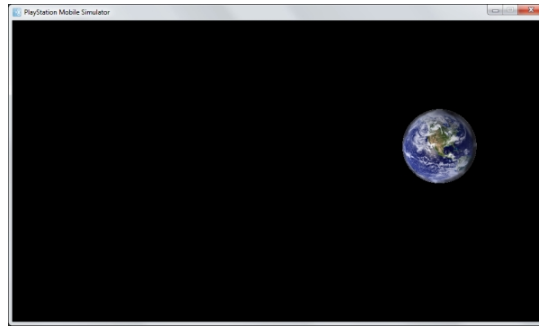
1. Open `AppMain.cs` and replace the existing code with the following code:

```
using System;
using
System.Collections.Generic;

using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Environment;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.Core.Input;

using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba

namespace Ch4_Example3
{
  public class AppMain
```

```
  {
    public static void Main
(string[] args)
    {
      Director.Initialize();
      Scene scene = new
Scene();

scene.Camera.SetViewFromViewport();

      SpriteUV earth1 = new
SpriteUV(new TextureInfo("/
Application/earth.png"));
      earth1.Scale =
earth1.TextureInfo.TextureSizef;
      earth1.Position = new
Vector2(0,250);


      Sequence sequence =
new Sequence();
      sequence.Add(new
DelayTime(3.0f));

      sequence.Add(new
MoveTo(
        new
Vector2(Director.Instance.GL.Context.GetV

250),10.0f));

      sequence.Add (new
MoveBy(new Vector2(
        -
Director.Instance.GL.Context.GetViewport(
- earth1.Scale.X/2,0)
          ,5.0f));

      var origColor =
earth1.Color;
```

```
            sequence.Add (new
        TintTo(new
        Vector4(0.0f,0.0f,0.0f,0.0f),10.0f));
            sequence.Add (new
        TintTo(origColor,10.0f));

            var origSize =
        earth1.Scale;
            sequence.Add (new
        ScaleTo(new
        Vector2(3.0f,3.0f),10.0f));
            sequence.Add (new
        CallFunc(()=>{
              earth1.Scale =
        origSize;
              earth1.Position =
        new Vector2(0,250);
            }));
            RepeatForever repeater
        = new RepeatForever();
            repeater.InnerAction =
        sequence;

        ActionManager.Instance.AddAction(repeater
            repeater.Run();

        scene.AddChild(earth1);

        Director.Instance.RunWithScene(scene);
          }
        }
        }
```

2. Now run the code by hitting *F5* and you should see the following output:



The following sequence of actions should occur:

- Waits for 3 seconds
- Moves the earth to the far right-hand side of the screen
- Moves back to the center of the screen
- Fades over 10 seconds to nothingness
- Fades back to original color over a period of 10 seconds
- Shrinks in 10 seconds to a size of 3 x 3 pixels
- Resized it back to its original size
- Reset position to the starting point and start the process over again

How it works...

We start off with the typical code, initializing the director, creating, and setting up a scene. We then load, size, and position a `SpriteUV` object with our earth graphic.

Next, we create a series of actions. The first is a `Sequence` action, which is used to hold a series of actions to be executed in the order they are added. The first action we add to the sequence is a `DelayTime` action, which simply waits for 3 seconds. Next, we add a `MoveTo` action, which moves the targeted node to the specified coordinate (the far right-hand side of the screen) over the specified period of time, in this case 10 seconds.

Next, we add a `MoveBy` action, which instead of moving to a specific coordinate, moves relative to a certain amount. In this case, we want to move by half the screen width minus half the sprite width, which will result in it being centered on the screen. This action

will also take 10 seconds.

Next, we take a copy of the current sprite color, then apply a `TintTo` action, passing in a `Vector4` object holding the target red, green, blue, and alpha values we wish to `TintTo` (in this case, black and transparent), and once again let it take 10 seconds. Next we `TintTo` back to the original color, causing our sprite to slowly reappear.

Then, we copy the sprites size, and scale it down to 3 x 3 pixels using the `ScaleTo` function. After 10 seconds, the sprite will have been completely shrunken; we then restore it to its previous size using the `CallFunc` action, which simply calls the passed in function, in this case a simple lambda that resizes and positions the sprite.

We then create one more action, a `RepeatForever` action, which simply causes the passed in action to repeat forever. We then set the action to be repeated using the `InnerAction` property. Finally, we call `ActionManager` and register our container action (`repeater`) and bind it to the earth node with a call to `AddAction()`. This call is how the individual actions know what node to apply their action against. As you can see, when you are dealing with a hierarchy of actions like a `Sequence` or `RepeatForever` action (or in this case a `RepeatForever` action containing a `Sequence` action), you only add the outermost action to the `ActionManager` singleton.

Finally, we tell the `repeater` action to start running, add our earth sprite to the scene, and then start the scene running with a call to `RunWithScene()`. It is of critical importance that you add an action to the `ActionManager` before calling `Run()` or you will generate an exception.

There's more...

The GameEngine2D library provides the following actions:

- `MoveBy`
- `MoveTo`

- `SkewBy`
- `SkewTo`
- `RotateBy`
- `RotateTo`
- `ScaleBy`
- `ScaleTo`
- `TintBy`
- `TintTo`
- `DelayTime`
- `CallFunc`
- `Repeat`
- `RepeatForver`
- `Sequence`

See also

- See the *Working with the ActionManager object* recipe for details on creating your own action classes

Transitioning between scenes

In this recipe, we are going to demonstrate the various graphical transitions you can perform when switching between scenes.

Getting ready

Load up PlayStation Mobile Studio and create a new project. Add a reference to `Sce.PlayStation.GameEngine2D`. The complete source for this example can be found in `Ch4_Example4`.

How to do it...

1. Open `AppMain.cs` and replace the existing code with the following code:

```
using System;
using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.Core.Imaging;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba

namespace Ch4_Example4
{
  public class ExampleScene
: Scene
  {
    String _text;
    TextureInfo _ti;
    Texture2D _texture;

    public ExampleScene
(string text, ImageColor
```

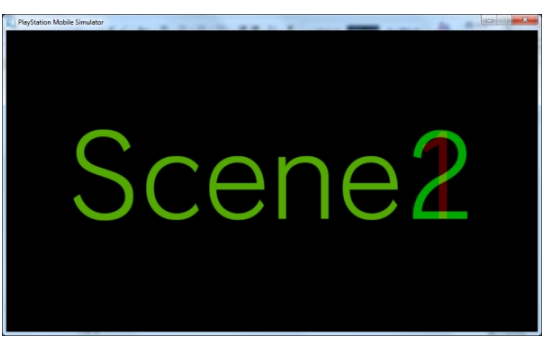
```
textColor, int textSize)
    {
      _text = text;
      var screenSize =
Director.Instance.GL.Context.GetViewport(
      var textImage = new
Image(ImageMode.Rgba,
                        new
ImageSize(screenSize.Width,screenSize.Hei
                         new
ImageColor(0,0,0,0));
      textImage.Decode();

      Font font = new
Font(FontAlias.System,textSize,FontStyle.


textImage.DrawText(text,
                textColor,
                new
Font(FontAlias.System,textSize,FontStyle.
                new
ImagePosition(
          (screenSize.Width
-
font.GetTextWidth(text))/2,

screenSize.Height/2 -
textSize/2));

      _texture = new
Texture2D(screenSize.Width,screenSize.Hei
false, PixelFormat.Rgba);

_texture.SetPixels(0,textImage.ToBuffer()
      textImage.Dispose();

      _ti = new
TextureInfo(_texture);
```

```
      SpriteUV sprite = new
SpriteUV(_ti);
      sprite.Pivot = new
Vector2(0.5f,0.5f);
      sprite.Position = new
Vector2(screenSize.Width/2,screenSize.Hei
      sprite.Scale =
_ti.TextureSizef;

this.Camera.SetViewFromViewport();
      this.AddChild(sprite);


this.RegisterDisposeOnExitRecursive();
    }

    public override void
OnEnter ()
    {
      DelayTime delay = new
DelayTime(3);
      CallFunc callFunc =
new CallFunc(()=> {
        switch(_text)
        {
          case "Scene1":

Director.Instance.ReplaceScene(
              new
TransitionCrossFade(
                new
ExampleScene("Scene2",new
ImageColor(0,255,0,255),220))
                  {
Duration=3.0f }
            );
            break;

          case "Scene2":
```

```
Director.Instance.ReplaceScene(
                new
TransitionDirectionalFade(
                  new
ExampleScene("Scene3",new
ImageColor(0,0,255,255),220))
                    {
Duration=3.0f,
Direction=Sce.PlayStation.HighLevel.GameE
}
              );
              break;
            case "Scene3":

Director.Instance.ReplaceScene(

                new
TransitionSolidFade(
                  new
ExampleScene("Scene4",new
ImageColor(255,255,255,255),220))
                    {
Duration=3.0f }
              );
              break;

          case "Scene4":

Director.Instance.ReplaceScene(
                new
TransitionCrossFade(
                  new
ExampleScene("Scene1",new
ImageColor(255,0,0,255),220))
                    {
                    Tween =
(x) =>
Sce.PlayStation.HighLevel.GameEngine2D.Ba

                    }
```

```
                );
                break;
            default:
                break;
        }
    });
    Sequence seq = new
Sequence();
    seq.Add(delay);
    seq.Add (callFunc);

ActionManager.Instance.AddAction(seq,this
    seq.Run ();
  }


  ~ExampleScene()
  {
    _ti.Dispose();
    _texture.Dispose();
  }
 }
}
```

2. Now run the code by hitting *F5* and you should see the following output:



The text **Scene1** will be displayed, then transition to **Scene2** then **Scene3** each in different colors, each change using a different graphical transition effect.

How it works...

`AppMain.cs` contains a minimal amount of code. First, we initialize the `Director` object, and create a new scene of our custom class `ExampleScene`. The `ExampleScene` class takes a few parameters: the text to display, the color to display the text in as well as the font size to use. Finally, we start the scene with a call to `RunWithScene()`.

The `ExampleScene` class starts off by declaring some member variables, the text to display on screen, the `Texture2D` object to draw the texture on, and `TextureInfo` to hold our texture details. Next, we store the passed in text and the screen dimensions. We then create a new clear black `Image` the same size as the screen. We then draw the text centered to the image, taking into account the width of the string. We then allocate the Texture2D and copy the image pixel data into it using `SetPixels()`. We are now done with the image, so we call `Dispose()` on it. Next, we create a `SpriteUV` object from our newly created texture, scale and then position it centered. Then, we add the sprite to the scene using `AddChild()`. Finally, we tell the scene that we want it to be disposed when `OnExit` events occur with a call to `RegisterDisposeOnExitRecursive()`.

The `OnEnter()` method is called when the scene is run (after the `RunWithScene()` call). We create a pair of actions, one is a time delay of 3 seconds and the other is a `CallFunc` action. In the function, we have a switch to the currently displayed text. If the current text is `Scene1`, we replace our scene with a new `ExampleScene` using a `TransitionCrossFade`, setting the text to `Scene2`. If the value is `Scene2`, we instead transition using a `TransitionDirectionalFade` object. If it is `Scene3`, we use `TransitionSolid` fade and finally in the event of `Scene4` we use a `TransitionCrossFade`, but this time instead of passing a `Duration` value, we pass in a `Tween` function that will control the fade.

We then create a `Sequence` action, add our delay and `CallFunc` actions to the sequence, and register it with `ActionManager` with a call to `AddAction`. Finally, we `Run()` the sequence.

Essentially the scene loads up and draws its text on screen in the specified color, then waits 3 seconds and begins transitioning the scene to a new scene with different text of a different color, using one of three different transition effects.

There's more...

The call to `RegisterDisposeOnExitRecursive()` was absolutely critical to keeping memory usage under control. Once you've called this method, when `OnExit` occurs, it causes the scene to call `Dispose` on each of its child nodes, then on itself. If you do not call this method when changing scenes, you can quickly run out of memory. There is another version, `RegisterDisposeOnExit`, but it will only dispose of the single value you pass as a parameter. If you are planning to reactivate the scene immediately (such as after showing a menu), you won't necessarily want to call this method.

You may have never encountered the word *tween* before now. It is a common expression in animation and is short hand for *in between*. It is a function that handles the transition between two states. In this recipe, we used a tween to control the rate that a fade transition occurred.

See also

- See the *"Hello World" drawing text on an image* recipe in Chapter 1, *Getting Started* for more details on text rendering

Simple collision detection

In this recipe we are going to demonstrate the basic collision detection facilities provided by the `Bounds2` class.

Getting ready

Load up PlayStation Mobile Studio and create a new project. Add a reference to `Sce.PlayStation.GameEngine2D`. You also need an image file; I am reusing our earth sprite from earlier recipes. The complete source for this example can be found in `Ch4_Example5`.

How to do it...

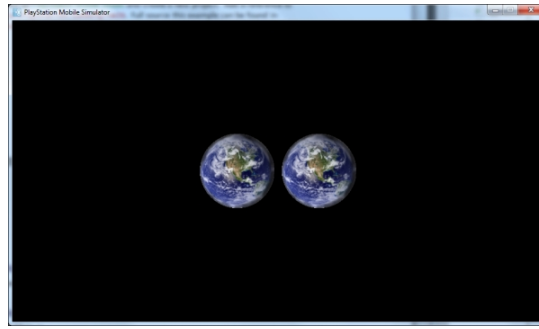
1. Open `AppMain.cs` and replace the existing code with the following code:

```
using System;
using
System.Collections.Generic;

using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Environment;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.Core.Input;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba

namespace Ch4_Example5
{
  public class AppMain{
    public static void Main
(string[] args){
```

```
      Director.Initialize();

      Scene scene = new
Scene();

scene.Camera.SetViewFromViewport();

      SpriteUV world1 = new
SpriteUV(new TextureInfo("/
Application/earth.png"));
      SpriteUV world2 = new
SpriteUV(new TextureInfo("/
Application/earth.png"));
      world1.Scale =
world2.Scale =
world1.TextureInfo.TextureSizef;
      world1.Pivot =
world2.Pivot = new
Vector2(0.5f,0.5f);
      world1.Position = new
Vector2(0,Director.Instance.GL.Context.Ge
      world2.Position = new
Vector2(Director.Instance.GL.Context.GetV


Director.Instance.GL.Context.GetViewport(


scene.AddChild(world1);

scene.AddChild(world2);


world1.Schedule((dt)=>{
        world1.Position =
new
Vector2(world1.Position.X
+1,world1.Position.Y);
      },0);
```

```
World2.Schedule((dt)=>{
        world2.Position =
new
Vector2(world2.Position.X-1,world2.Positi

        Bounds2 w1bounds =
new Bounds2();
        Bounds2 w2bounds =
new Bounds2();

world1.GetContentWorldBounds(ref
w1bounds);

world2.GetContentWorldBounds(ref
w2bounds);

if(w1bounds.Overlaps(w2bounds))
        {
          world1.Position =
new
Vector2(0,Director.Instance.GL.Context.Ge
          world2.Position =
new
Vector2(Director.Instance.GL.Context.GetV


Director.Instance.GL.Context.GetViewport(
        }
      },0);

Director.Instance.RunWithScene(scene);
    }
  }
}
```

2. Now hit *F5* to run the application and you should see the following output:



The two earth sprites will move towards the center of the screen until they collide, then they reset back to their original position and start the process all over again.

How it works...

We initialize the `Director` object, create, and configure a scene. We then create two `SpriteUV` object's using the same `earth.png` image; we position one on the left-center portion of the screen, and the other on the right-center portion of the screen. We then add both sprites to the scene.

We schedule an update routine on the first sprite, `world1`, to increase the *x* position by *1* per update. We then schedule *world2* to decrease its *x* position by *1* each update. Additionally, it will check for a collision each frame. First it creates a pair of `Bounds2` objects, `w1bounds` and `w2bounds`. It then gets the current bounding box of each sprite by calling `GetContentWorldBounds()`, with the results stored in `w1bounds`/`w2bounds`. We then check to see if `w1bounds` overlaps with `w2bounds`, and if it does we reset both sprites back to their starting positions.

Finally, we start the scene running with a call to `RunWithScene()`.

There's more...

In addition to bounding boxes, you can also use a sphere (`Sphere2`), plane (`Plane2`), and convex polygon (`ConvexPoly2`). In addition to checking for overlaps, you can also use `Bounds2` to

check if a point is inside the bounds, get the intersection between two bounds (returned as another bounds), and more. There is also a complete 2D physics system if you require more complex collision behavior, which we will cover in a later chapter.

You may be wondering where the pixel level collision detection is, often referred to as pixel perfect collision detection. Short answer, there isn't any. If you loaded all your textures from an `Image` class, it would be possible to store the alpha values of every pixel. There, is however, no way to access the image data of any class other than `Image`. This is due to performance; direct pixel access is quite slow.

See also

- See the *Querying if a collision occurred* and *Rigid body collision shapes* recipes in Chapter 5, *Working with Physics2D* for more details on handling collisions using the physics system
- See the *Manipulating a texture's pixels* recipe in Chapter 3, *Graphics with GameEngine2D* for details on directly accessing the alpha channel, if you want to implement pixel level collision detection

Playing background music

In this recipe we are going to illustrate how to add background music to your game.

Getting ready

Load up PlayStation Mobile Studio and create a new project. Add a reference to `Sce.PlayStation.GameEngine2D`. This example also requires a pair of MP3 files; I used two public domain classical music scores. The complete source for this example can be found in `Ch4_Example6`.

How to do it...

1. Add the two MP3 files to your project; set their **Build Action** to **Content**.
2. Open `AppMain.cs` and replace the existing code with the following code:

```
using System;
using
System.Collections.Generic;
using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Environment;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.Core.Input;
using
Sce.PlayStation.Core.Audio;
using
Sce.PlayStation.HighLevel.GameEngine2D;

namespace Ch4_Example6
{
  public class AppMain{
```

```
    private static
Dictionary<string,Bgm>
_songs;
    private static BgmPlayer
_bgmPlayer;

    private static void
PlaySong(string filename){
      if(_bgmPlayer != null)
{
        if(_bgmPlayer.Status
== BgmStatus.Playing)
          _bgmPlayer.Stop();

_bgmPlayer.Dispose();
      }
      if(!
_songs.ContainsKey(filename))

_songs.Add(filename,new
Bgm(filename));

      _bgmPlayer =
_songs[filename].CreatePlayer();
      _bgmPlayer.Play();
    }


    public static void Main
(string[] args){
      Director.Initialize();
      Scene scene = new
Scene();
      _songs = new
Dictionary<string, Bgm>();

      CallFunc playSong1 =
new CallFunc(
        ()=>  PlaySong ("/
```

```
Application/Bach_Fugue.mp3")
        );

      CallFunc playSong2 =
new CallFunc(
        ()=>  PlaySong ("/
Application/
RideOfTheValkyries.mp3")
        );
      Sequence seq = new
Sequence();
      seq.Add(playSong1);
      seq.Add (new
DelayTime(30));
      seq.Add(playSong2);


ActionManager.Instance.AddAction(seq,scer
      seq.Run();


Director.Instance.RunWithScene(scene);

    }
  }
}
```

3. Hit the *F5* key to run the application. A window will open and Bach's *Fugue in G Minor* will start playing. Thirty seconds later it will be replaced by Wagner's *Ride of the Valkyries*.

How it works...

We start by declaring a pair of member variables, a dictionary of `Bgm` (**Bgm** = **Background music**) files, indexed by their filenames, and a `BgmPlayer`. We then create a function `PlaySong()` that plays an opened file and plays the MP3 file passed in. First, it checks to see if `_bgmPlayer` has been allocated, if it has, we check to see if it is playing and, if it is, we stop playback with a call

to `_bgmPlayer.Stop()`. We then dispose off this player.

Next we check to see if there is an entry in our dictionary already for a song by this name; if there isn't, we create a new `Bgm` object and add it to our dictionary. Next, we get the `Bgm` file from the dictionary and create a new `BgmPlayer` by calling `CreatePlayer()`. Finally, we play the song with a call to `Play()`.

In the main function, we initialize the `Director` object, create a scene, and allocate our `_songs` dictionary. We then create a pair of `CallFunc` actions, the first to call `PlaySong()` to play the Bach piece, the second to call `PlaySong()` to play the Wagner piece. Next, we declare a sequence, and add our two function actions to it, with a 30 second `DelayTime` action in between. Next, we register our `Sequence` action and associate it with the scene. Then, we run the sequence and then start the scene running.

There's more...

`Bgm` only supports MP3 files with the following specifications:

- `Codec`: MPEG Layer3
- `Num channels`: 1/2
- `Sampling rate`: 44100/48000 Hz
- `Bitrate`: 128 to 320 kbps

If you need to convert your audio to MP3, I highly recommend you to check out the freely available open source application *Audacity*, which you can download from http://bit.ly/NRDuky. Audacity is available for Windows, Linux, and MacOS, and is capable of an amazing number of audio processing and conversion tasks.

In addition to playing and stopping a song, you can also pause, resume, change the volume, loop, and change position and playback rate using the `BgmPlayer` class. There are no events to determine when a song is finished, so you will need to poll the current song's status if you want, when it is done.

See also

- See the *Using predefined actions* recipe for details on the `Sequence` and `CallFunc` actions

Playing sound effects

In this recipe we are going to play back some WAV sound effects.

Getting ready

Load up PlayStation Mobile Studio and create a new project. Add a reference to `Sce.PlayStation.GameEngine2D`. You will need a pair of WAV files to play as sound effects. The complete source for this example can be found in `Ch4_Example7`.

How to do it...

1. Add the two WAV files to your project and set their **Build Action** to **Content**.
2. Open **AppMain.cs** and replace the existing code with the following code:

```
using System;
using
System.Collections.Generic;

using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Environment;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.Core.Input;
using
Sce.PlayStation.Core.Audio;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba
namespace Ch4_Example7
{
  public class AppMain
```

```
  {
    private static Sound
_machinegunSound;
    private static Sound
_explosionSound;
    private static
SoundPlayer
_soundPlayerMachinegun;
    private static
SoundPlayer
_soundPlayerExplosion;

    public static void Main
(string[] args)  {
      Director.Initialize();

Director.Instance.RunWithScene(new
Scene(),true);

      _machinegunSound = new
Sound("Application/
machinegun.wav");
            _explosionSound
= new Sound("Application/
explosion.wav");

      _soundPlayerMachinegun
=
_machinegunSound.CreatePlayer();
      _soundPlayerExplosion
=
_explosionSound.CreatePlayer();

      bool quit = false;

      while(!quit)
      {

Director.Instance.Update();
```

```
            if(Input2.GamePad0.Cross.Down)
                quit = true;


            if(Input2.GamePad0.Up.Down)
                {

            _soundPlayerMachinegun.Play();
                }


            if(Input2.GamePad0.Down.Down)
                {

            _soundPlayerExplosion.Play();
                }


            Director.Instance.Render();

            Director.Instance.GL.Context.SwapBuffers(

            Director.Instance.PostSwap();
                }

            _machinegunSound.Dispose();

            _explosionSound.Dispose();

            _soundPlayerMachinegun.Dispose();

            _soundPlayerExplosion.Dispose();
            }
        }
    }
```

3. Now hit *F5* to run the application. Press the up arrow to play a machine gun sound effect, and press the down arrow to play an explosion sound. Finally, press the *X*

button (*S* in the simulator) to exit.

How it works...

We start off by initializing the `Director` object (getting used to this yet?), and start running with `RunWithScene()`, passing in a new `Scene` and `true`, indicating that we will manually manage the event loop this time.

We then load each of our sound effect files by passing each file path to the Sound constructor. For each `Sound` we then create a `SoundPlayer` by calling `CreatePlayer()`.

We then loop until the bool `quit` is `true`. Since we are manually handling the loop, there are four mandatory functions we need to call. Additionally, after the `Update()` call, we check to see if the Cross (**X**) button is down, in which case we set `quit` to `true`, exiting the loop. If **Up** is down (ok, that felt weird) we play our machine gun sound by calling its `Play()` method. We perform the same action on `_soundPlayerExplosion` if the down key is down.

We then call the remaining mandatory methods for the game loop. Finally, both `SoundPlayer` and `Sound` are `IDisposable`, so we call the `Dispose()` method on each object.

There's more...

Like `Bgm`, sound effects are played by first creating a `Sound` object and passing a WAV file to it, then creating a player from it. A player will only allow playback of a single instance of a sound effect at a time, so if you call play again, it will simply start the sound effect over. If you want to play multiple copies of the same sound effect at once, you are going to need to create multiple players. You can of course have multiple SoundPlayer functions playing at once.

The `SoundPlayer` class includes properties for controlling the volume, panning, looping, and playback rate. You can also check the status (playing or stopped) of a sound effect. Like `Bgm`, you need to poll if you want to determine when a sound effect has

finished playing.

`SoundPlayer` supports WAV file playback with the following specifications:

- `Num channels`: 1/2
- `Sampling rate`: 22050/44100 Hz
- `Sampling bit depth`: 16 bit only
- `* Linear` PCM only

> **Note**
>
> The sound effects used in this recipe were downloaded from http://FreeSound.org, a gigantic archive of free sound files. You do need to sign up before you can download any sounds. If you are looking for sound files, it is an excellent place to start.
>
> Sony also provides a number of royalty free sound effects available in the **SFX Asset Library**. They can be downloaded on the PSM Developer Portal. Once you have logged in to the portal they are available for download at https://psm.playstation.net/static/general/dev/en/sdk_assets.html. This link will not work until you log in.

See also

- See the *A game loop, GameEngine2D style* recipe in Chapter 3, *Graphics with GameEngine2D* for more details on manually handling the game loop

Chapter 5. Working with Physics2D

In this chapter we will cover:

- Creating a simple simulation with gravity
- Switching between dynamic and kinematic
- Creating a (physics!) joint
- Applying force and picking a physics scene object
- Querying if a collision occurred
- Rigid body collision shapes
- Building and using an external library

Introduction

Physics is an integral part of many modern games, including many 2D games. The massive success of *Angry Birds* and similar titles really brought physics to the masses. Fortunately, Sony included a 2D library in the PlayStation Mobile SDK, **Physics2D**. In this chapter, we are going to look at a number of ways this library can be used.

Creating a simple simulation with gravity

In this recipe, we are going to configure a simple physics scene consisting of a single rigid body (a sphere) and the effects of gravity on it.

Getting ready

Load up PlayStation Mobile Studio and create a new project. Add references to `Sce.PlayStation.HighLevel.GameEngine2D` and `Sce.PlayStation.HighLevel.Physics2D`. We also need to add a circular sprite; I used a picture of the earth aptly named `earth.png`. The complete source and all images used can be found in `Ch5_Example1`.

How to do it...

1. Add the earth sprite to your project, and set its **Build Action** to **Content**.
2. Open up the `AppMain.cs` file and replace the existing code with the following code:

```
using System;
usingSce.PlayStation.Core;
usingSce.PlayStation.Core.Graphics;
usingSce.PlayStation.Core.Input;

using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba
using
Sce.PlayStation.HighLevel.Physics2D;

namespace Ch5_Example1 {
  public class AppMain {
```

```
    private static
TextureInfo _ti;
    private static Texture2D
_texture;

    public static void Main
(string[] args)  {
      Director.Initialize();

      Scene scene = new
Scene();

scene.Camera.SetViewFromViewport();

      _texture = new
Texture2D("/Application/
earth.png",false);
      _ti = new
TextureInfo(_texture);

      SpriteUV sprite = new
SpriteUV(_ti);
      sprite.Scale =
_ti.TextureSizef;
      sprite.Pivot = new
Vector2(0.5f,0.5f);
      sprite.Position = new
Vector2(

Director.Instance.GL.Context.GetViewport(

Director.Instance.GL.Context.GetViewport(
- sprite.Scale.Y/2);


scene.AddChild(sprite);
```

```
PhysicsScenephysicsScene =
new PhysicsScene();

physicsScene.InitScene();
      physicsScene.sceneName
= "Test";
      physicsScene.NumBody =
1;
      physicsScene.NumShape
= 1;


physicsScene.sceneShapes[0]
= new
PhysicsShape(sprite.Scale.Y/2.0f);

physicsScene.sceneBodies[0]
= new
PhysicsBody(physicsScene.sceneShapes[0],1

physicsScene.sceneBodies[0].position
= sprite.Position;

physicsScene.sceneBodies[0].rotation
= 0;

physicsScene.sceneBodies[0].shapeIndex
= 0;

      physicsScene.gravity =
new Vector2(0,-98.0f);



Director.Instance.RunWithScene(scene,
true);

      bool done = false;
      while(!done){
```

```
Director.Instance.Update();

        Vector2
touchPosition =
Input2.Touch.GetData(0)
[0].Pos;
        Vector2 diffPosition
= new Vector2();
physicsScene.Simulate(-1,ref
touchPosition, ref
diffPosition);

        sprite.Position =
physicsScene.SceneBodies[0].Position;


Director.Instance.Render();

Director.Instance.GL.Context.SwapBuffers(

Director.Instance.PostSwap();
        if(sprite.Position.Y
+ sprite.Scale.Y/2 < 0)
          done=true;
      }


physicsScene.ReleaseScene();
      _ti.Dispose();
      _texture.Dispose();
    }
  }
}
```

How it works...

This code starts off initializing the `Director` singleton, creating a new scene, and setting the camera. It then creates a new `SpriteUV` object to hold our `earth.png` texture. We then position

the texture at the top-middle of the screen and finally add it to the scene.

Next, we create the `PhysicsScene` object, which is completely unrelated to GameEngine2D's `Scene` class. Start off by initializing the scene with a call to `InitScene` and name it by setting the property `SceneName` (the name has no importance). We then tell our scene how many `PhysicsBody` and `PhysicsShape` objects it is going to contain.

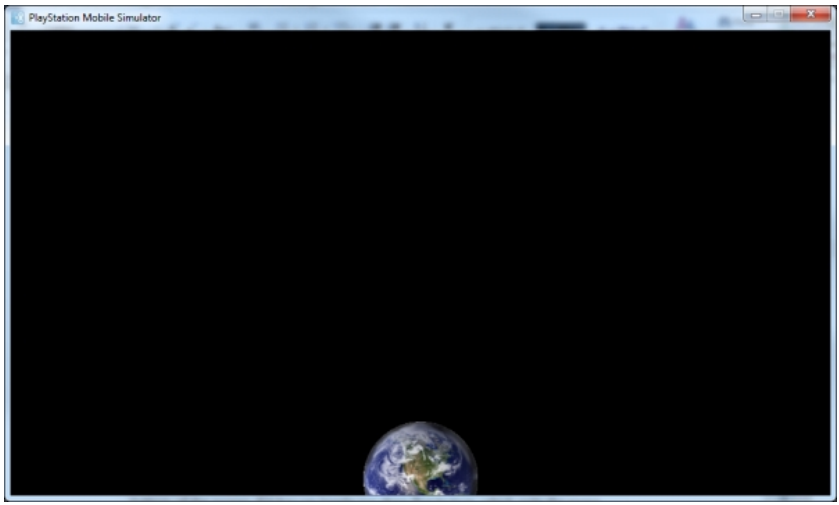
We then declare our shape in the array `physicsScene.SceneShapes`. In the `PhysicsShape` constructor, we pass a single parameter, which is the radius of the sphere's rigid body that is half the width of our sprite

Next, we declare our `PhysicsBody` constructor in the array `physicsScene.sceneBodies`. We pass in the `PhysicsShape` constructor we just created, as well as the default mass to use in the simulation. We then set our `PhysicsBody` position to match our earth sprite, set the rotation to 0 and the `shapeIndex` to 0. The shape index is the location of the `PhysicsShape` this `PhysicsBody` is going to occupy in the `sceneShapes` array. It is possible for multiple `PhysicsBody` objects to use the same shape.

Now we set the scene gravity to (0.0,-98.0f), which results in the earth being pulled down by an acceleration starting at 98 pixels per second. Finally we start the scene running with a call to `Director.RunWithScene`, passing false in addition to the scene, indicating we will be manually handling the event loop.

In the `event` loop, we call the `Director` object's `Update` method, then declare a pair of `Vector2` objects, `touchPosition` and `diffPosition`, which the `Simulate` scene requires, although they serve no purpose in this example. After calling `Simulate`, we then synchronize the sprite's position to match the newly updated `SceneBody` object's position. We then call the required `Render`, `SwapBuffer`, and `PostSwap` methods. We then check to see if the sprite has passed to the bottom of the screen. If it has, we toggle our done flag to true, which exits the game loop. Finally, we clean up before the program exits.

Now press *F5* to run your game and you will see the following output:



The earth sprite will appear at the top-middle of the screen and then gravity will pull it to the bottom, which will cause the program to exit.

There's more...

It is important to realize what a physics engine does and does not do. A physics engine runs a physical simulation on a series of physics bodies (shapes, joints), using factors such as mass, force, gravity, and more, to determine movement in a (often) realistic manner. You define the objects in your physics scene, the properties of the scene itself, and then update it each frame with a call to `PhysicsScene.Simulate`, which causes the simulation to advance in time calculating how all of its bodies interact and move. These movements, however, are not displayed in your GameEngine2D scene. It is your responsibility to take the updated position data from the physics simulation and update your game's graphics accordingly.

In this particular recipe, we made one physics unit equal to one pixel, which is why setting gravity to (0,-98) resulted in the earth moving starting at 98 pixels per second (and increasing as it falls). In a more complex scene, this is not the ideal approach to take, as it

quickly makes your physics engine numbers meaningless. Internally, the Physics2D engine treats *1.0* mass as *1* kg, while 1 unit in Physics2D is a meter. We will deal with mapping between Physics2D units and GameEngine2D pixels in a later recipe.

Note

`PhsyicsScene` contains three very important pre-allocated arrays, `SceneBodies`, `SceneShapes`, and `SceneJoints`. The `SceneBodies` array contains 250 items, while `SceneShapes` and `SceneJoints` are both 100 items each. These represent the maximum number of rigid bodies, shapes, and joints you can have in a single `PhysicsScene` array. You use the values `NumBody`, `NumShape`, and `NumJoint` to tell Physics2D how many of each you use. It is very important you set each of these values correctly, or a crash will occur.

If you are asking yourself, *Why did Sony choose to use pre-allocated arrays instead of just lists?*, that is a very good question! It's much better question than my answer; I have no idea.

See also

- See the *Adding a sprite to a scene* recipe in Chapter 3, *Graphics with GameEngine2D* for more information on drawing sprites with GameEngine2D

Switching between dynamic and kinematic

In this recipe, we are going to demonstrate toggling a physics body between dynamic and kinematic. We are also going to illustrate the effect of the coefficient of restitution on the physics scene.

Getting ready

Load up PlayStation Mobile Studio and create a new project. Add a reference to `Sce.PlayStation.HighLevel.GameEngine2D` and `Sce.PlayStation.HighLevel.Physics2D`. I will again be using our `earth.png` sprite; add this image or a similar one to your scene and set its **Build Type** to **Content**. The complete source and all images used can be found in `Ch5_Example2`.

How to do it...

1. Open `AppMain.cs` and edit it as follows:

```
using System;
usingSystem.Collections.Generic;

usingSce.PlayStation.Core;
usingSce.PlayStation.Core.Environment;
usingSce.PlayStation.Core.Graphics;
usingSce.PlayStation.Core.Input;

using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba
using
Sce.PlayStation.HighLevel.Physics2D;

namespace Ch5_Example2
{
```

```
  public class AppMain
  {

    private static
TextureInfo _ti;
    private static Texture2D
_texture;

    public static void Main
(string[] args)
    {
      Director.Initialize();

      Scene scene = new
Scene();

scene.Camera.SetViewFromViewport();

      _texture = new
Texture2D("/Application/
earth.png",false);
      _ti = new
TextureInfo(_texture);

      SpriteUV sprite = new
SpriteUV(_ti);

      sprite.Scale =
_ti.TextureSizef;
      sprite.Pivot = new
Vector2(0.5f,0.5f);
      sprite.Position = new
Vector2(

Director.Instance.GL.Context.GetViewport(

Director.Instance.GL.Context.GetViewport(
- sprite.Scale.Y/2);
```

```
scene.AddChild(sprite);



PhysicsScenephysicsScene =
new PhysicsScene();

physicsScene.InitScene();
      physicsScene.SceneName
= "Demo scene";
      physicsScene.NumBody =
2;
      physicsScene.NumShape
= 2;
      physicsScene.SceneMax
= new
Vector2(1000.0f,5000.0f);

physicsScene.RestitutionCoeff
= 1.0f;


physicsScene.SceneShapes[0]
= new
PhysicsShape(sprite.Scale.Y/2.0f);

physicsScene.SceneBodies[0]
= new
PhysicsBody(physicsScene.sceneShapes[0],1

physicsScene.SceneBodies[0].Position
= sprite.Position;

physicsScene.SceneBodies[0].Rotation
= 0;

physicsScene.SceneBodies[0].ShapeIndex
= 0;

physicsScene.SceneBodies[0].ColFriction
```

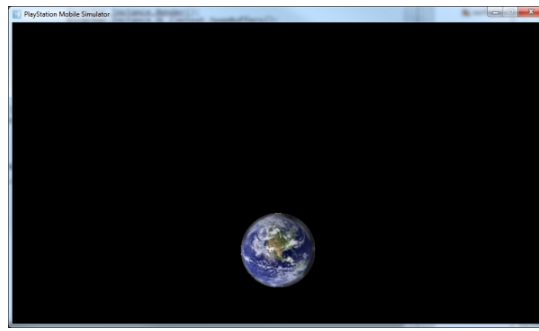
```
= 0.01f;

physicsScene.SceneBodies[0].SetBodyKinema



physicsScene.SceneShapes[1]
= new PhysicsShape(new
Vector2(

Director.Instance.GL.Context.GetViewport(
        1.0f));


physicsScene.SceneBodies[1]
= new
PhysicsBody(physicsScene.SceneShapes[0],P

physicsScene.SceneBodies[1].position
= new Vector2(0,0);

physicsScene.SceneBodies[1].shapeIndex
= 1;


physicsScene.SceneBodies[1].SetBodyStatic


      physicsScene.Gravity =
new Vector2(0.0f,-98.0f);



Director.Instance.RunWithScene(scene,
true);

      bool done = false;
      while(!done)
      {
```

```
Director.Instance.Update();

System.Diagnostics.Debug.WriteLine(physic


if(Input2.GamePad0.Down.Down
== true)

physicsScene.sceneBodies[0].BackToDynamic

if(Input2.GamePad0.Up.Down
== true)

physicsScene.sceneBodies[0].SetBodyKinema

if(Input2.GamePad0.Left.Press
== true)

physicsScene.RestitutionCoeff
=
physicsScene.RestitutionCoeff
-0.1f;

if(Input2.GamePad0.Right.Press
== true)

physicsScene.RestitutionCoeff
=
physicsScene.RestitutionCoeff
+0.1f;

if(Input2.GamePad0.Cross.Down
== true)
          done = true;


if(physicsScene.CheckOutOfBound(physicsSc
          done = true;

        Vector2
```

```
touchPosition =
Input2.Touch.GetData(0)
[0].Pos;
        Vector2 diffPosition
= new Vector2();

physicsScene.Simulate(-1,ref
touchPosition, ref
diffPosition);

        sprite.Position =
physicsScene.SceneBodies[0].Position;
        if(sprite.Position.Y
+ sprite.Scale.Y/2 < 0 ||

(System.Math.Floor
(sprite.Position.Y) == 0 +
sprite.Scale.Y/2 )

&&physicsScene.SceneBodies[0].Velocity.Y<
1.0f)
            done=true;



Director.Instance.Render();

Director.Instance.GL.Context.SwapBuffers(

Director.Instance.PostSwap();


      }


physicsScene.ReleaseScene();
      _ti.Dispose();
      _texture.Dispose();
    }
  }
```

```
}
```

2. Hit the *F5* key to run your application and you should see the following output:



The earth sprite will appear at the top of the screen. Pressing the down arrow will toggle the object to dynamic, causing gravity to take effect and the earth to start falling. Pressing up arrow again will toggle back to kinematic. Pressing the left arrow will decrease the coefficient of restitution while pressing the right arrow will increase it. The higher the value, the more the earth will *bounce*. When it hits the bottom of the screen, it will bounce up as determined by the coefficient amount. Once it comes to a complete stop, or you press *X*, the game will exit.

How it works...

We start off with the usual initialization code, initialize the `Director` object, set and configure the scene, load our earth sprite, position it at the top-middle of the screen, and finally add it to the scene.

The `PhysicsScene` setup is very similar to the prior recipe. This time we have two `PhysicsShape` objects and two `PhysicsBody` objects in our `scene` array, so we set `NumBody` and `NumShape` accordingly. `SceneMax` is the outer boundary for physics objects (once reached, the scene will stop processing it). Since we are allowing our earth to bounce a substantial amount offscreen, we greatly increase the `SceneMax` object's `Y` value. By default, `SceneMax` is normally set to (1000,1000) while `SceneMin` is

(-1000,-1000). Finally, we set the scene's `RestitutionCoeff` to `1.0f`.

Next, we create a `PhysicsShape` and `PhysicsBody` to represent our earth sprite, set to a radius half the sprite's width. Using this newly created shape, we create a new `PhysicsBody` object, giving it a mass of `100.0f`. We match the rigid body position to the sprite position, set `Rotation` to `0`, `ShapeIndex` to `0` (matching the value we passed in the constructor), its `ColFriction` to `0.01f` (meaning virtually no friction on collision), and initially set the body to kinematic by calling the function `SetBodyKinematic`.

Next, we create another `PhysicsShape` object; this one is box shaped. We pass the width and height of the box in to the `PhysicsShape` constructor. We are creating a 1 pixel high box across the bottom of the screen for the earth to bounce off. In the `PhysicsBody` constructor we pass in the newly created `PhysicsShape` object, as well as the value `PhysicsUtility.FltMax` for mass. `PhysicsUtility.FltMax` is the highest value mass can go up to and can effectively be considered to be infinity. Finally, since this body is completely stationary, we set it to static with the call `SetBodyStatic`.

We then set the gravity to `-98.0f` in the *Y* direction and run our GameEngine2D scene using `RunWithScene`, passing true, indicating we will be manually handling our `event` loop.

In the `event` loop, we call the mandatory `Update` method. We print out the earth's physics body location to the debug window since it can spend a lot of time offscreen. You can see this value in the **Application Output** window in Studio and is a handy way to debug offscreen physics objects.

Next we check the gamepad status. If the user presses the down arrow on the d-pad, we switch the earth to **Dynamic**. If the user presses the up arrow, the earth is set to **Kinematic**. If the left d-pad is pressed, we decrease the `RestitutionCoeff` by `0.1f`, while pressing the right one increases it by `0.1f`. Finally, pressing *X* will exit the application.

Next, we check to see if the earth has managed to bounce out of

bounds (a *Y* value greater than 5000) with a call to `CheckOutOfBound` and exit if it has. Next, we create two (unused) `Vector2` values required by `Simulate` and then call `Simulate`, which causes the physics scene to update and advance in time. Then we update the GameEngine2D sprite's position to match the physics body location.

Then we check to make sure that the earth hasn't somehow gone off the bottom of the screen and that the velocity hasn't dropped below `1.0f`. You can't check for 0, since in a physics simulation values will almost never be pure zero. If the earth has stopped moving, we exit the application. We then call the mandatory `Render`, `SwapBuffers`, and `PostSwap` calls. Once the game loop ends, we clean up before exiting.

There's more...

The `RestitutionCoeff` variable holds the **restitution of coefficient**, which represents the ratio of speeds before and after a collision. A value of `1.0f`, all other values being equal, means that the rigid body will bounce out with the same force it came in at. A value of `0.0f` will mean no bounce at all, while a value greater than `1.0f` will cause a much greater amount of recoil. In the real world, however, this value would never exceed `1.0f`, so if you want a realistic simulation, never set this value higher than `1`. Unlike other physics engines, in Physics2D the restitution coefficient is set at the scene level instead of the body level. By default in our example with the value of `1.0f`, the earth will bounce back to pretty much where it started.

You may have noticed when we switched the object from Dynamic to Kinematic it no longer fell. When it is kinematic, external forces (collisions, gravity, air friction, and so on) will have no effect on the movement of the body. Only direct influences, such as velocity, will affect its movement. A static body is even less affected by the physics simulation and basically only exists for other bodies to interact with. It is, however, much lighter weight and easier for the simulation to process, so if you have an object that is stationary, you should set it to static.

In addition to setting a rigid body to kinematic or static to save on processing power in the physics simulation, you can also put inactive (offscreen, invisible, and so on) bodies to sleep. Simply set the sleep property to true and the physics engine will skip it during the Simulate process.

See also

- See the *Handling the controller's d-pad and buttons* recipe in Chapter 2, *Controlling Your PlayStation Mobile Device* for more details on using the d-pad

Creating a (physics!) joint

In this recipe, we are going to look at simulating a joint to create a pendulum style effect, swinging one physics body around another.

Getting ready

Load up PlayStation Mobile Studio and create a new project. Add a reference to `Sce.PlayStation.HighLevel.GameEngine2D` and `Sce.PlayStation.HighLevel.Physics2D`. We will be re-using our earth sprite from the earlier recipe, this time reduced in size by 50 percent and named, appropriately enough, `earthSmall.png`. Add this image or a similar one to your scene and set its **Build Type** to **Content**. The complete source and all images used can be found in `Ch5_Example3`.

How to do it...

1. Open `AppMain.cs` and change the code to the following:

```
using System;
usingSce.PlayStation.Core;
usingSce.PlayStation.Core.Graphics;
usingSce.PlayStation.Core.Input;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba
using
Sce.PlayStation.HighLevel.Physics2D;

namespace Ch5_Example3 {
  public class AppMain {

    private static
TextureInfo _ti;
    private static Texture2D
```

```
_texture;

    public static void Main
(string[] args) {
      Director.Initialize();

      Scene scene = new
Scene();

scene.Camera.SetViewFromViewport();

      _texture = new
Texture2D("/Application/
earthSmall.png",false);
      _ti = new
TextureInfo(_texture);

      SpriteUV sprite = new
SpriteUV(_ti);
      sprite.Scale =
_ti.TextureSizef;
      sprite.Pivot = new
Vector2(0.5f,0.5f);
      sprite.Position = new
Vector2(

Director.Instance.GL.Context.GetViewport(

Director.Instance.GL.Context.GetViewport(
- sprite.Scale.Y/2);

      SpriteUV sprite2 = new
SpriteUV(_ti);
      sprite2.Scale =
_ti.TextureSizef;
      sprite2.Pivot = new
Vector2(0.5f,0.5f);
      sprite2.Position = new
Vector2(
        0 +
```

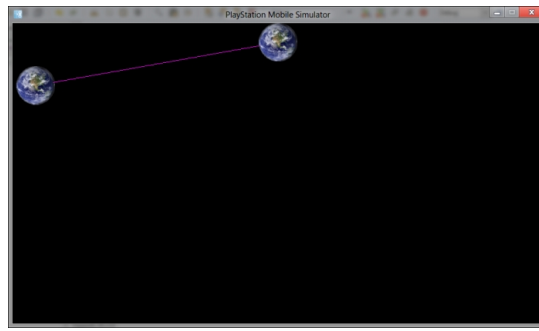
```
sprite2.Scale.X/2,

Director.Instance.GL.Context.GetViewport(
- sprite2.Scale.Y/2);

scene.AddChild(sprite);

scene.AddChild(sprite2);

      scene.AdHocDraw += ()
=> {

Director.Instance.DrawHelpers.DrawLineSeg
      };


PhysicsScenephysicsScene =
new PhysicsScene();

physicsScene.InitScene();
      physicsScene.SceneName
= "Demo scene";
      physicsScene.NumBody =
2;
      physicsScene.NumShape
= 1;
      physicsScene.NumJoint
= 1;



physicsScene.SceneShapes[0]
= new
PhysicsShape(sprite.Scale.Y/2.0f);

physicsScene.SceneBodies[0]
= new
PhysicsBody(physicsScene.sceneShapes[0],1

physicsScene.SceneBodies[0].Position
```

```
= sprite.Position;

physicsScene.SceneBodies[0].Rotation
= 0;

physicsScene.SceneBodies[0].ShapeIndex
= 0;

physicsScene.SceneBodies[0].SetBodyStatic


physicsScene.SceneBodies[1]
= new
PhysicsBody(physicsScene.sceneShapes[0],1

physicsScene.SceneBodies[1].Position
= sprite2.Position;

physicsScene.SceneBodies[1].Rotation
= 0;

physicsScene.SceneBodies[1].ShapeIndex
= 0;



physicsScene.SceneJoints[0]
= new
PhysicsJoint(physicsScene.SceneBodies[0],

physicsScene.SceneBodies[1],
      sprite.Position,


0,1);

physicsScene.SceneJoints[0].AngleLim
=0;

      physicsScene.Gravity =
```

```
new Vector2 (0.0f, -98.0f);

Director.Instance.RunWithScene(scene,
true);

      bool done = false;
      while(!done) {

Director.Instance.Update();


if(Input2.GamePad0.Cross.Down
== true)
          done = true;
        Vector2
touchPosition =
Input2.Touch.GetData(0)
[0].Pos;
        Vector2 diffPosition
= new Vector2();

physicsScene.Simulate(-1,ref
touchPosition, ref
diffPosition);

        sprite.Position =
physicsScene.SceneBodies[0].Position;
        sprite2.Position =
physicsScene.SceneBodies[1].Position;


Director.Instance.Render();

Director.Instance.GL.Context.SwapBuffers(

Director.Instance.PostSwap();


      }
```

```
            physicsScene.ReleaseScene();
                _ti.Dispose();
                _texture.Dispose();
            }
        }
    }
```

2. Hit the *F5* key to run your application and you should see the following output:



One small earth sprite located in the top-left corner of the screen will swing in an arc around another earth sprite locate at the top-center of the screen. The joint connecting them will be drawn using a purple line. Once it completes its arc, it will then proceed to return in the other direction and loop until the user hits the **X** button or *S* key to exit.

How it works...

This code starts off with initialization code nearly identical to that of the last few recipes. We create and configure our scene, load our `earthSmall.png` texture into a pair of `SpriteUV` objects, and position one at the top-center of the screen and the other at the top-left corner of the screen. We then add both sprites to the scene. Next we register an `AdHocDraw` method to draw a line between the two sprites, representing the joint.

Next we set up the physics scene; it will contain one `PhysicsShape`, two `PhysicsBody` objects, and one

`PhysicsJoint`, so we set `NumBody`, `NumShape`, and `NumJoint` accordingly. We configure a spherical shape for our two earths, and then create a `PhysicsBody` for each one. The center one is not going to be moving at all, so we create it as static with the call `SetBodyStatic`.

Next, we create a `PhysicsJoint` in the `SceneJoints` array. The `PhysicsJoint` constructor takes the two rigid bodies the joint is going to connect, the joint's pivot point (our centered earth sprite's position), then the index in the `SceneBodies` array of the two `PhysicsBodies` the joint hinges. Yes, like with `ShapeIndex`, these two indexes seem redundant to me as well. Next, we set the joint's angle limit to *0* using the `AngleLim` property. A value of 0 puts no limitations on the rotation of the joint.

Next, we set the scene's gravity to `-98.0f` on the y axis, which is what is responsible for the motion in our scene. Finally, we tell the GameEngine2D scene to start running with a call to `Director.Instance.RunWithScene`. In our game loop, we call the various required methods, update the physics simulation, and then update the position of our sprites (somewhat redundantly in the case of the centered sprite, since it is static). Finally, we exit if the user presses the **X** button or *S* key in the Simulator.

There's more...

Joints can be constrained to a single axis, creating a joint that only moves in a single direction. For example, to simulate the movement of a sliding door, you could constrain movement to the x axis only with the following code:

```
sceneJoints[index].axis1Lim = new
Vector2(1.0f, 0.0f);
sceneJoints[index].axis2Lim = new
Vector2(0.0f, 0.0f);
```

To allow this joint to move along both axes again, you would set `axis2Lim` to `(0.0f,1.0f)`.

See also

- See the *Creating scenes* recipe in Chapter 3, *Graphics with GameEngine2D* for more details about using `AdHocDraw`

Applying force and picking a physics scene object

In this recipe, we are going to cover two separate concepts at once. First, we are going to look at how to apply force to a rigid body. Then we are going to look at how to handle selecting objects within the physics scene by clicking/touching.

Getting ready

Load up PlayStation Mobile Studio and create a new project. Add a reference to `Sce.PlayStation.HighLevel.GameEngine2D` and `Sce.PlayStation.HighLevel.Physics2D`. Once again we will be making use of our small earth sprite, `earthSmall.png`. The complete source and all images used can be found in `Ch5_Example4`.

How to do it...

1. Open `AppMain.cs` and enter the following code:

```
using System;
usingSce.PlayStation.Core;
usingSce.PlayStation.Core.Graphics;
usingSce.PlayStation.Core.Input;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba
using
Sce.PlayStation.HighLevel.Physics2D;

namespace Ch5_Example4
{
  public class AppMain
  {
```

```
    private static
TextureInfo _ti;
    private static Texture2D
_texture;

    privateconst float PtoM
= 50.0f;
    static float
_screenWidth;
    static float
_screenHeight;

    static Vector2
GetClickPos(Vector2 point) {

      // Convert from NDC (
-1 to 1 ) to normalized
coordinates ( 0 to 1 )
      varnewx =
(_screenWidth/2 + (point.X *
_screenWidth/2))/
_screenWidth;
      varnewy =
(_screenHeight/2 + (point.Y
* _screenHeight/2))/
_screenHeight;

      // Convert from
normalized to screen
coordinates at a ratio of
Pixels to Meters
      // PtoM = 50 means 50
pixels to a meter
      return new
Vector2((newx *
_screenWidth) / PtoM, (newy
* _screenHeight) / PtoM);
        }

    public static void Main
```

```
(string[] args) {
      Director.Initialize();

      Scene scene = new
Scene();

scene.Camera.SetViewFromViewport();
      _screenWidth =
Director.Instance.GL.Context.GetViewport(
      _screenHeight =
Director.Instance.GL.Context.GetViewport(
      _texture = new
Texture2D("/Application/
earthSmall.png",false);
      _ti = new
TextureInfo(_texture);

      SpriteUV sprite = new
SpriteUV(_ti);
      sprite.Scale =
_ti.TextureSizef;
      sprite.Pivot = new
Vector2(0.5f,0.5f);
      sprite.Position = new
Vector2(
        _screenWidth/2,
        _screenHeight/2);


scene.AddChild(sprite);


PhysicsScenephysicsScene =
new PhysicsScene();

physicsScene.InitScene();
      physicsScene.SceneName
= "Demo scene";
      physicsScene.NumBody =
1;
```

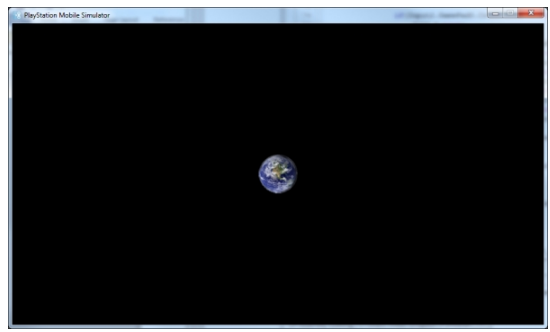
```
      physicsScene.NumShape
= 1;
      physicsScene.SceneMin
= new Vector2(
        0.0f -
(sprite.Scale.X/2)/PtoM,
        0.0f -
(sprite.Scale.Y/2)/PtoM);

      physicsScene.SceneMax
= new Vector2(
        (_screenWidth +
sprite.Scale.X/2)/PtoM,
        (_screenHeight +
sprite.Scale.Y/2)/PtoM);

      physicsScene.Gravity =
new Vector2(0.0f,0.0f);


physicsScene.SceneShapes[0]
= new
PhysicsShape((sprite.Scale.Y/2.0f)/
PtoM);

physicsScene.SceneBodies[0]
= new
PhysicsBody(physicsScene.sceneShapes[0],1

physicsScene.SceneBodies[0].Position
= sprite.Position/PtoM;


physicsScene.SceneBodies[0].Rotation
= 0;

physicsScene.SceneBodies[0].ShapeIndex
= 0;
```

```
Director.Instance.RunWithScene(scene,
true);

      bool done = false;
      Vector2 touchPosition
= new Vector2();
      Vector2 diffPosition =
new Vector2();


      while(!done) {

Director.Instance.Update();


if(Input2.GamePad0.Cross.Down
== true)
          done = true;

        varsceneBody =
physicsScene.SceneBodies[0];

if(Input2.GamePad0.Right.Press
== true)
          sceneBody.Force =
sceneBody.Force.Add(new
Vector2(40.0f,0.0f));

if(Input2.GamePad0.Left.Press
== true)
          sceneBody.Force =
sceneBody.Force.Subtract(new
Vector2(40.0f,0.0f));

if(Input2.GamePad0.Up.Press
== true)
          sceneBody.Force =
sceneBody.Force.Add(new
Vector2(0.0f,40.0f));
```

```
if(Input2.GamePad0.Down.Press
== true)
          sceneBody.Force =
sceneBody.Force.Subtract(new
Vector2(0.0f,40.0f));


if(physicsScene.CheckOutOfBound(sceneBody
        done = true;

        touchPosition =
GetClickPos(Input2.Touch00.Pos);
        vartouchIndex = -1;

if(Input2.Touch00.Press)
        {
          touchIndex =
physicsScene.CheckPicking(ref
touchPosition,refdiffPosition,true);
          if(touchIndex !=
-1)
            done = true;
        }


physicsScene.Simulate(-1,ref
touchPosition, ref
diffPosition);
        sprite.Position =
physicsScene.SceneBodies[0].Position
* PtoM;


Director.Instance.Render();

Director.Instance.GL.Context.SwapBuffers(

Director.Instance.PostSwap();
      }
```

```
            physicsScene.ReleaseScene();
                _ti.Dispose();
                _texture.Dispose();
            }
        }
    }
```

2. Press *F5* to run the application and you will see the following output:



3. Press the d-pad or arrow key in any direction and force in that direction will be applied. Each time you press in a single direction more force will be applied and the earth sprite will move faster. Adding force in the opposite direction will decrease speed and eventually reverse the direction of travel.
4. Hitting the **X** button or *S* key will exit the application.
5. Finally, if you click on the earth or leave the screen, the application will exit.

How it works...

In addition to our usual `Texture2D` and `TextureInfo` member variables, we declare `_screenWidth` and `_screenHeight` to cut down a bit on code length. We are also defining a constant `float` value, `PtoM`. This stands for *Pixels To Meters* and is used to map from screen space to physics space. This will be explained in more detail shortly.

Next, we define a function `GetClickPos`, which will translate a click into a physics scene coordinate. This looks rather confusing because there are three sets of conversions going on here. First our touch coordinate, passed in as a point, is in a normalized device coordinate format, which ranges in value from (-1,-1) to (1,1). To translate that coordinate to normalized coordinates (0,0) to (1,1), you multiply the *x* value by *half the screen width + half the screen width divided by the screen width*. The same process is now repeated for the *y* coordinate. We then translate this coordinate into GameEngine2D coordinates by multiplying *x* by the screen width and *y* by the screen height. We finally convert it to Physics2D coordinate space by dividing it by our `PtoM` ratio.

Within the `main` function, the code is very similar to the prior recipes. We create and configure the scene, cache the dimensions for convenience, load our texture into a `SpriteUV` that we center to the screen, and then add the sprite to the scene.

Next, we setup our `PhysicsScene`. It contains a single `PhysicsShape` object and a single `PhysicsBody` object, the collision sphere around our sprite. The only major difference from previous recipes is that we modify all values by converting pixels to meters. Next, we turn gravity off by setting it to a `Vector2` variable with a value of (`0.0`,`0.0`). Then, we set the boundaries for our physics simulation. We want enough room for our screen, plus half of the width of our earth sprite, since the earth sprite has the pivot set to its middle. Once again, this value needs to be converted to Physics2D coordinates. Next, we construct our `PhysicsShape` and `PhysicsBody` values, giving it a mass of 10 kg, a very light planet. Feel free to update this value, but if you do, you are going to have to add a lot more force later on! After running the scene we declare a pair of `Vector2` values, `touchPosition` and `diffPosition`, which we are actually going to use in this recipe.

In our game loop, in addition to the four required method calls, we check the status of the gamepad. If the user hits the **X** button (or *S* key) we exit the application. If the user hits left, we subtract 40 force along the x axis, while if the user presses down we subtract 40 force along the y axis. If the user presses right or up we instead increase the force by 40. All of these values are cumulative, so pushing left

twice will move faster, while pushing left and then right will cancel movement out completely. We then check if our rigid body has left the simulation boundaries with a call to `CheckOutOfBounds`, and exit if it has.

Next, we handle user touches (or clicks on the simulator). We pass the value of `Input2.Touch00.Pos` in to the `GetClickPos` function we defined earlier, which translates it to Physics2D compatible coordinates. We then check to see if an actual touch has occurred, and if one has, we check to see if there is something in the simulation at that location. This is done with the call `CheckPicking`, which takes as parameters the touch location, a `Vector2` variable that will hold the distance between the center of the rigid body and the touch position, and a bool indicating how thorough we want the click testing to be. If there is nothing at the touched location, `CheckPicking` will return *-1*. Otherwise, it will return the index of the body (within the `SceneBodies` array) of the object that was selected. If the user successfully touched the earth, we exit the application.

Next, we update the simulation with a call to `Simulate`. Once the simulation has updated, we update our sprite's position to match its corresponding rigid body position. This time we need to map from physics coordinates to screen coordinates, so we multiply our coordinates by `PtoM`.

There's more...

Up until this recipe, we have paid no attention to the actual units we were using; we simply treated one unit in the physics simulation to equal one pixel in the game. There is nothing wrong with this, but it can lead to some remarkably odd values. In this recipe, we convert from pixels to physics units and vice versa. Physics2D is set up so that *1 unit of measurement = 1 meter*, while *1 unit of weight = 1 kg* and *1 unit of force = 1 Newton*. When you start using real world units, the simulation becomes much easier to understand. In this case we are going to translate at a rate of 50 to 1, in that *50 pixels = 1 meter*. This value was chosen arbitrarily; feel free to change it.

You may at this point be wondering about the values you pass in to

simulate. The first is the index of the item the user clicked (within the `SceneBodies` array), while `clickPosition` and `diffPosition` allow you to specify two points used to apply velocity to the rigid body specified by the `clickIndex` variable. So basically it allows you to apply motion to the item touched and, frankly, I think it was bad design.

Earlier we mentioned that force was applied as Newtons. Therefore, when we subtract 40 force, we are actually subtracting 40 Newtons. One Newton is equivalent to the force of earth's gravity acting on an object weighing about 100 gms, which is approximately the weight of an apple. Had Newton been hit on the head with say, an anvil, a Newton would be a much different unit of measure!

See also

- See the *Using the Input2 wrapper class* recipe in Chapter 2, *Controlling Your PlayStation Mobile Device* for more information on `Input2` and coordinate conversion

Querying if a collision occurred

In this recipe, we are going to demonstrate how to check for a collision between rigid bodies.

Getting ready

Load up PlayStation Mobile Studio and create a new project. Add a reference to `Sce.PlayStation.HighLevel.GameEngine2D` and `Sce.PlayStation.HighLevel.Physics2D`. This time we will be using our full-sized earth sprite `earth.png`. The complete source and all images used can be found in `Ch5_Example5`.

How to do it...

1. Replace the existing code in `AppMain.cs` with the following code:

```
using System;
usingSce.PlayStation.Core;
usingSce.PlayStation.Core.Graphics;
usingSce.PlayStation.Core.Input;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba
using
Sce.PlayStation.HighLevel.Physics2D;

namespace Ch5_Example5 {
  public class AppMain {

    private static Texture2D
_texture;
    private static
TextureInfo _ti;
    privateconst float PtoM
= 50.0f;
```

```
    public static void Main
(string[] args) {
      Director.Initialize();

      Scene scene = new
Scene();

scene.Camera.SetViewFromViewport();

      _texture = new
Texture2D("/Application/
earth.png",false);
      _ti = new
TextureInfo(_texture);

      SpriteUV sprite = new
SpriteUV(_ti);
      sprite.Scale =
_ti.TextureSizef;
      sprite.Pivot = new
Vector2(0.5f,0.5f);
      sprite.Position = new
Vector2(
        sprite.Scale.X/2,

Director.Instance.GL.Context.GetViewport(

      SpriteUV sprite2 = new
SpriteUV(_ti);
      sprite2.Scale =
_ti.TextureSizef;
      sprite2.Pivot = new
Vector2(0.5f,0.5f);
      sprite2.Position = new
Vector2(

Director.Instance.GL.Context.GetViewport(
- sprite.Scale.X/2,
```

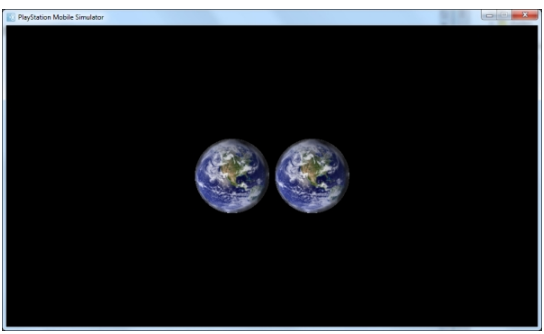
```
Director.Instance.GL.Context.GetViewport(


scene.AddChild(sprite);

scene.AddChild(sprite2);


PhysicsScenephysicsScene =
new PhysicsScene();

physicsScene.InitScene();
      physicsScene.SceneName
= "Demo scene";
      physicsScene.NumBody =
2;
      physicsScene.NumShape
= 1;
      physicsScene.Gravity =
new Vector2(0.0f,0.0f);


physicsScene.SceneShapes[0]
= new
PhysicsShape((sprite.Scale.Y/2.0f)/
PtoM);


physicsScene.SceneBodies[0]
= new
PhysicsBody(physicsScene.sceneShapes[0],1

physicsScene.SceneBodies[0].Position
= sprite.Position/PtoM;

physicsScene.SceneBodies[0].Rotation
= 0;

physicsScene.SceneBodies[0].ShapeIndex
= 0;
```

```
physicsScene.SceneBodies[1]
= new
PhysicsBody(physicsScene.sceneShapes[0],1

physicsScene.SceneBodies[1].Position
= sprite2.Position/PtoM;

physicsScene.SceneBodies[1].Rotation
= 0;

physicsScene.SceneBodies[1].ShapeIndex
= 0;


physicsScene.SceneBodies[0].Force
= new Vector2(200.0f,0);

physicsScene.SceneBodies[1].Force
= new Vector2(-200.0f,0);


Director.Instance.RunWithScene(scene,
true);

      Vector2 touchPosition
= new Vector2();
      Vector2 touchDiff =
new Vector2();

      bool quit = false;
      while(!quit)
      {

Director.Instance.Update();

if(Input2.GamePad0.Cross.Press)
          quit = true;
```

```
physicsScene.Simulate(-1,ref
touchPosition,reftouchDiff);
        sprite.Position =
physicsScene.SceneBodies[0].Position
* PtoM;
        sprite2.Position =
physicsScene.SceneBodies[1].Position
* PtoM;


if(physicsScene.QueryContact(0,1))
        {

physicsScene.SceneBodies[0].Position
= new Vector2(

sprite.Scale.X/2,

Director.Instance.GL.Context.GetViewport(
/PtoM;

physicsScene.SceneBodies[0].Force
= new Vector2(200.0f,0);


physicsScene.SceneBodies[1].Position
= new Vector2(

Director.Instance.GL.Context.GetViewport(
- sprite.Scale.X/2,

Director.Instance.GL.Context.GetViewport(
/PtoM;

physicsScene.SceneBodies[1].Force
= new Vector2(-200.0f,0);
        }

Director.Instance.Render();
```

```
            Director.Instance.GL.Context.SwapBuffers(

            Director.Instance.PostSwap();
                }
                _ti.Dispose();
                _texture.Dispose();
                Director.Terminate();
            }
        }
    }
```

2. Now press *F5* to run your application and you should see the following output:



Two earth sprites will start centered, one on the left and the other on the right-hand side of the screen. They will both travel towards the screen's middle until a collision occurs, at which point they will be reset to their starting positions and the process will begin again. Pressing the **X** button or **S** key will exit the application.

How it works...

Almost all of the code is repeated from previous recipes. We create our scene, load, and position the two sprites at each side of the screen, vertically centered, then add them both to the scene. We then configure the physics scene to contain two rigid bodies and one shape. Like the prior recipe, we will be converting to physics coordinates using a ratio of 50 pixels to a meter. We then add a force of *200N* to each body, sending them on a collision course at

the middle of the screen.

In our game loop, in addition to the mandatory function calls, we update the physics simulation with a call to `Simulate`, and then update each sprite's position accordingly. We then check to see if a collision occurred. This is done with the `QueryContact` method. You pass `QueryContact` the index of the two bodies with the `SceneBodies` array that you want to check for a collision between. If a collision occurs, `QueryContact` returns `true`, otherwise it returns `false`. In the event a collision occurs, we re-center our two sprites and reset their force values (the collision will have greatly reduced the force of each).

There's more...

In addition to setting a rigid body to kinematic, dynamic, or static, you can also set it to be a trigger by calling `SetBodyTrigger`. If a body is a trigger, it will not participate in the physics simulation, but it can receive collisions. You can use the method `QueryContactTrigger` to check if a body collided with the trigger. You pass `QueryContactTrigger` the index of the trigger itself, and an array of body indexes to check for collisions.

See also

- See the *Switching between dynamic and kinematic* recipe for more details on the effects of each setting

Rigid body collision shapes

In this recipe, we are going to look at the three different shapes a rigid body can have, the sphere, box, and convex hull.

Getting ready

Load up PlayStation Mobile Studio and create a new project. Add a reference to `Sce.PlayStation.HighLevel.GameEngine2D` and `Sce.PlayStation.HighLevel.Physics2D`. We are going to re-use our small earth sprite, `earthSmall.png`, once again, as well as our trusty `FA-18H.png` sprite to provide a slightly more complex shape. The complete source and all images used can be found in `Ch5_Example6`.

How to do it...

1. Open `AppMain.cs` and replace the existing code with the following code:

```
using System;
usingSce.PlayStation.Core;
usingSce.PlayStation.Core.Graphics;
usingSce.PlayStation.Core.Input;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Ba
using
Sce.PlayStation.HighLevel.Physics2D;

namespace Ch5_Example6 {
  public class AppMain {
    private static Texture2D
_textureJet;
    private static
TextureInfo _tiJet;
```

```
    private static Texture2D
_textureEarth;
    private static
TextureInfo _tiEarth;

    privateconst float PtoM
= 50.0f;

    public static void Main
(string[] args) {
      Director.Initialize();
      Scene scene = new
Scene();

scene.Camera.SetViewFromViewport();

      SpriteUVjet,earth;
      _textureJet = new
Texture2D("/Application/
FA-18H.png",false);
      _tiJet = new
TextureInfo(_textureJet);
      jet = new
SpriteUV(_tiJet);
      jet.Position = new
Vector2(Director.Instance.GL.Context.GetV

Director.Instance.GL.Context.GetViewport(
      jet.Pivot = new
Vector2(0.5f,0.5f);
      jet.Scale =
_tiJet.TextureSizef;


      _textureEarth = new
Texture2D("/Application/
earthSmall.png",false);
      _tiEarth = new
TextureInfo(_textureEarth);
      earth = new
```

```
SpriteUV(_tiEarth);
      earth.Position = new
Vector2(Director.Instance.GL.Context.GetV

Director.Instance.GL.Context.GetViewport(
      earth.Pivot = new
Vector2(0.5f,0.5f);
      earth.Scale =
_tiEarth.TextureSizef;

      scene.AddChild(jet);
      scene.AddChild(earth);

      PhysicsSceneps = new
PhysicsScene();
      ps.InitScene();
      ps.SceneName = "Demo
Scene";
      ps.NumBody = 3;
      ps.NumShape = 3;

      Vector2[] points = new
Vector2[9];

      Vector2 center = new
Vector2(jet.Scale.X/2,jet.Scale.Y/2);
      points[0] = new
Vector2(0.0f,center.Y) /
PtoM;
      points[1] = new
Vector2(50.0f, 30.0f) /
PtoM;
      points[2] = new
Vector2(center.X -
10.0f,10.0f) / PtoM;
      points[3] = new
Vector2(center.X - 10.0f, -
30.0f) / PtoM;
      points[4] = new
Vector2(30.0f,-center.Y +
```

```
30.0f) / PtoM;
      points[5] = new
Vector2(-30.0f,-center.Y +
30.0f) / PtoM;
      points[6] = new
Vector2(-center.X +
10.0f,-30.0f) / PtoM;
      points[7] = new
Vector2(-center.X +
10.0f,10.0f) / PtoM;
      points[8] = new
Vector2(-50.0f, 30.0f) /
PtoM;
      //Jet
      ps.SceneShapes[0] =
new PhysicsShape(points,9);
      ps.SceneBodies[0] =
new
PhysicsBody(ps.SceneShapes[0],2.0f);

ps.SceneBodies[0].ShapeIndex
= 0;

ps.SceneBodies[0].Position =
new
Vector2(Director.Instance.GL.Context.GetV

Director.Instance.GL.Context.GetViewport(
/ PtoM;

      //Earth
      ps.SceneShapes[1] =
new
PhysicsShape(earth.Scale.X/2
/ PtoM);
      ps.SceneBodies[1] =
new
PhysicsBody(ps.SceneShapes[1],1.0f);

ps.SceneBodies[1].ShapeIndex
```

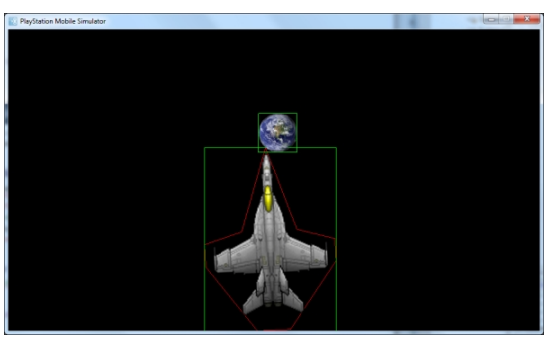
```
= 1;

ps.SceneBodies[1].Position =
new
Vector2(Director.Instance.GL.Context.GetV

Director.Instance.GL.Context.GetViewport(
/ PtoM;

      //Ground
      ps.SceneShapes[2] =
new PhysicsShape(
        new
Vector2(Director.Instance.GL.Context.GetV
/ PtoM);
      ps.SceneBodies[2] =
new
PhysicsBody(ps.SceneShapes[1],PhysicsUtil

ps.SceneBodies[2].Position =
new Vector2(0.0f,0.0f);

ps.SceneBodies[2].ShapeIndex
= 2;

      scene.AdHocDraw += ()
=> {
        //Cannot use
DrawConvexPolygon because
each point needs to be
converted back to pixel
space

Director.Instance.DrawHelpers.SetColor(ne
Vector4(255.0f,0.0f,0.0f,255.0f));
        for(inti=1; i< 10; i
++){
            Vector2 p1,p2;
            //Convert to
screen coordinates and
```

```
rotate to match physics
object
            if(i == 9) {
              p1 =
ps.SceneShapes[0].vertList[8];
              p2 =
ps.SceneShapes[0].vertList[0];
            }
            else {
              p1 =
ps.SceneShapes[0].vertList[i-1];
              p2 =
ps.SceneShapes[0].vertList[i];
            }
            p1 =
p1.Rotate(ps.sceneBodies[0].Rotation);
            p1 = p1  *PtoM +
jet.Position;


            p2 =
p2.Rotate(ps.SceneBodies[0].Rotation);
            p2 = p2 * PtoM +
jet.Position;


Director.Instance.DrawHelpers.DrawLineSeg
          }



Director.Instance.DrawHelpers.SetColor(ne
Vector4(0.0f,255.0f,0.0f,255.0f));
        var bounds = new
Bounds2(ps.SceneBodies[0].AabbMin*PtoM,ps

Director.Instance.DrawHelpers.DrawBounds2

        var bounds2 = new
Bounds2(ps.SceneBodies[1].AabbMin*PtoM,ps
```

```
Director.Instance.DrawHelpers.DrawBounds2
      };


Director.Instance.RunWithScene(scene,true

      Vector2 touchPosition
= new Vector2();
      Vector2 touchDiff =
new Vector2();

      bool quit = false;
      while(!quit) {

if(Input2.GamePad0.Cross.Press)
          quit = true;


Director.Instance.Update();
        ps.Simulate(-1,ref
touchPosition,reftouchDiff);

        jet.Position =
ps.sceneBodies[0].Position *
PtoM;
        jet.Rotation = new
Vector2((float)System.Math.Cos(ps.SceneBo

      (float)System.Math.Sin
(ps.SceneBodies[0].Rotation));

        earth.Position =
ps.sceneBodies[1].Position *
PtoM;
        earth.Rotation = new
Vector2((float)System.Math.Cos(ps.SceneBo

      (float)System.Math.Sin
(ps.SceneBodies[1].Rotation));
```

```
            Director.Instance.Render ();

            Director.Instance.GL.Context.SwapBuffers(

            Director.Instance.PostSwap();
                  }

                  _tiJet.Dispose();
                  _textureJet.Dispose();
                  _tiEarth.Dispose();

            _textureEarth.Dispose();

                  Director.Terminate();
               }
            }
         }
```

2. Press *F5* to run the application and you should see the following output:



The `F18` sprite will fall from the sky due to gravity, followed shortly by a small earth sprite. The two bodies will then collide and respond accordingly. The bounding box of each sprite is drawn in green, while the exact boundaries of the `F18` sprite are drawn in red. As you can see, the bounding box shrinks as the sprite rotates.

How it works...

The scene starts off typically, loading and positioning both of our sprites, and adding them to the scene. We then create a `PhysicsScene` constructor containing three `PhysicsBody` objects with three `PhysicsShape` objects. We start off creating the first shape, which is going to be made up of a series of `Vector2` objects composing the hull around our sprite. We construct these points at the scene origin, with positioning relative to the center of our sprite, like this:

Once again, we are using the 50 pixels to 1 meter ratio, so we divide each `Vector2` by `PtoM` to adjust it. We then construct our jet's `PhysicsShape` by passing the `Vertex2` array and array length in to the `PhysicsShape` constructor. This creates a polygon around the hull that will be used for collision detections. We also create a sphere physics body for our earth object and a box rigid body for the ground at the bottom of the screen.

Next, we register an `AdHocDraw` method to draw the lines composing the polygon around the jet. First we set the current

drawing color to red. Then we start at the second item, and draw from the previous `Vector2` in the array to the current item, unless it is the last item, in which case we draw from the final `Vector2` back to the first one, closing off the shape. As part of the `Simulate` process, the `SceneBody` controlling the jet will be rotated in addition to being moved, so we need to apply this rotation to each `Vector2` in the array.

We then need to translate each line from Physics2D coordinates relative to the origin to screen coordinates relative to the sprite's position. Finally, we draw the line on the screen using the helper method `DrawLineSegment`.

Once we have drawn the bounding polygon, we switch the color to green. We get the bounding box around our jet and earth bodies using the `AabbMin` and `AabbMax` values (**Aabb** = **Axis Aligned Bounding Box**). Then draw these bounds using the `DrawHelper` object's method `DrawBounds2`.

Within the game loop, we check for the user pressing **X**, in which case we exit the application. Otherwise we update the simulation, and then update the jet `Position` and `Rotation`, as well as the earth's `Position` and `Rotation`. Then we call the mandatory functions required if you manually manage your game loop. Finally, before the program exits, we clean up after ourselves.

There's more...

The `DrawHelpers` object contains a method named `DrawConvexPoly2` that could have drawn our shape in a single call. However, we needed to translate each line segment to use our screen coordinates, so we instead choose to draw the outline as a series of line segments. It should be noted that `DrawHelpers` is not meant for production code and has not been optimized.

Building and using an external library

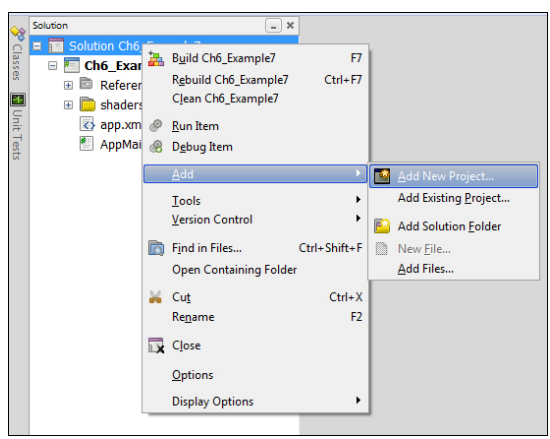
In this recipe, we will examine the process of building and using an external library.

Getting ready

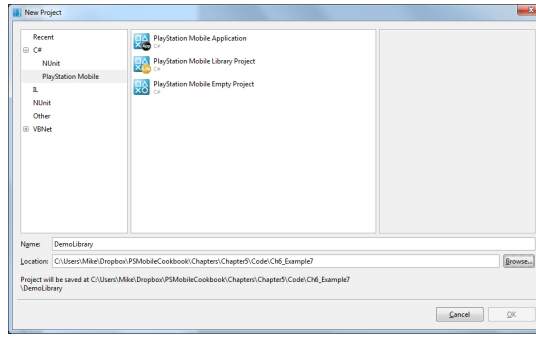
Load up PlayStation Mobile Studio. Open a solution you wish to add a library to, or create a new workspace.

How to do it...

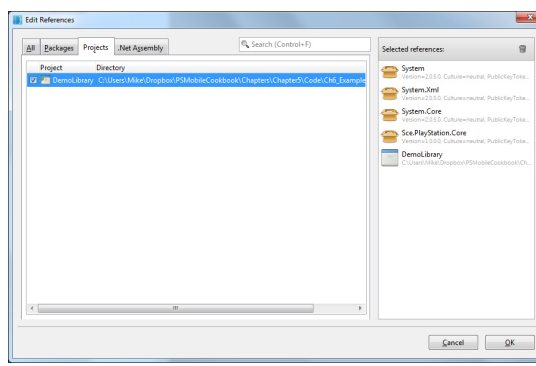
1. Right-click on the solution at the top of the **Solution panel** tree. In the context menu select **Add** | **Add New Project**.



2. In the resulting dialog box, on the left-hand side, expand **C#** and select **PlayStation Mobile**. On the right-hand side, select **PlayStation Mobile Library Project**. Select a location for your library, give it a name, and then click on **OK**.



3. In the **Solution** panel, your library should now be added. Now add code and resources from the library you wish to import.
4. Now, we will add the library to your existing project. Right-click your application's `References` folder in the **Solution** panel and select **Edit References…**.
5. In the resulting dialog box, select the **Projects** tab. Simply check the checkbox to the left of your library name and then press **OK**.



How it works...

There are two major limitations of external libraries you can use with PlayStation Mobile. First, they need to be compatible with the subset of the `.NET` runtime included with PlayStation Mobile. Generally, this shouldn't prove to be a big problem. Second, they need to include no unsafe or native code. Generally, if the library is simply a wrapper around a native DLL such as Box2D, it will not work with PlayStation Mobile.

There's more...

In addition to the included Physics2D engine, there are a few third party physics engines that will work with PlayStation Mobile Studio. The first is the **BEPU** physics engine (http://bit.ly/QbjQV9), while the second is the **FarSeer** physics engine (http://bit.ly/NY3U9T), both of which are fully open source and hosted on **CodePlex**.

There is also a port of the popular **MonoGame** library (http://bit.ly/QD43hh), a library designed to make porting of XNA games to various platforms (iOS, Android, Linux, Mac, and now PlayStation Mobiles) easier.

Chapter 6. Working with GUIs

In this chapter we will cover:

- "Hello World" – HighLevel.UI style
- Using the UI library within a GameEngine2D application
- Creating and using hierarchies of widgets
- Creating a UI visually using UIComposer
- Displaying a MessageBox dialog
- Handling touch gestures and using UI effects
- Handling language localization

Introduction

In this chapter we are going to learn about the `HighLevel.UI` toolkit as well as UIComposer, a visual UI building tool. When combined together they enable you to create rich user interfaces for your game or application with relative ease. The UI library ships with a rather extensive set of premade controls; we will take a closer look at a few of them.

"Hello World" – HighLevel.UI style

In this recipe we are going to create a simple "Hello World" application, implemented using a `Panel` and a `Label` widget from the `HighLevel.UI` library.

Getting ready

Load up **PlayStation Mobile Studio** and create a new project. Add a reference to `Sce.PlayStation.HighLevel.UI`. You can download this complete project as `Ch6_Example1`.

How to do it...

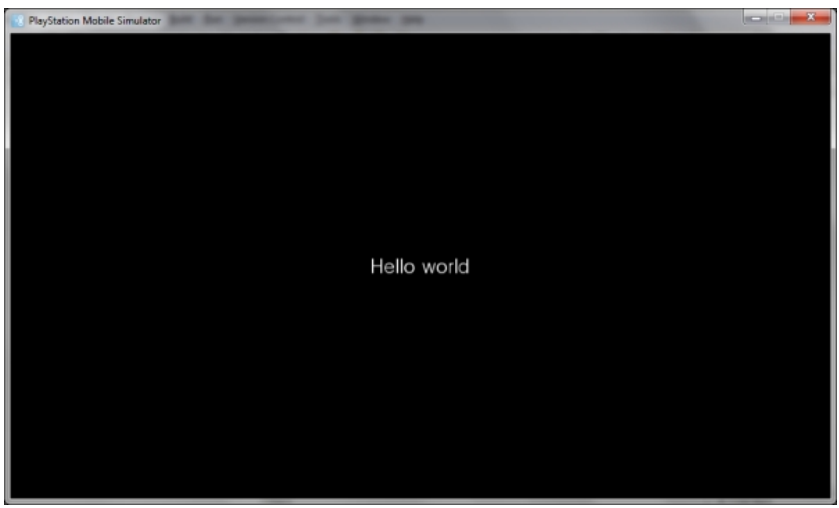
Edit the `AppMain.cs` file to match the following:

```
using System;
using Sce.PlayStation.Core;
using Sce.PlayStation.Core.Graphics;
using Sce.PlayStation.Core.Input;
using Sce.PlayStation.HighLevel.UI;


namespace Ch6_Example1
{
  public class AppMain {
    public static void Main() {
      GraphicsContext graphics = new
GraphicsContext();
      UISystem.Initialize(graphics);

      var scene = new
Sce.PlayStation.HighLevel.UI.Scene();
      var panel = new
Sce.PlayStation.HighLevel.UI.Panel();
```

```
      panel.Width =
graphics.Screen.Width;
      panel.Height =
graphics.Screen.Height;


      var label = new
Sce.PlayStation.HighLevel.UI.Label();
      label.HorizontalAlignment =
HorizontalAlignment.Center;
      label.VerticalAlignment =
VerticalAlignment.Middle;
      label.SetPosition (
        graphics.Screen.Width/2 -
label.Width/2,
        graphics.Screen.Height/2 -
label.Height/2
        );
      label.Text = "Hello world";

      panel.AddChildLast(label);

scene.RootWidget.AddChildLast(panel);

      UISystem.SetScene(scene);

graphics.SetViewport(0,0,graphics.Screen.Width,gr
      graphics.SetClearColor(new
Vector4(0,0,0,255));

      bool done = false;
      while(!done)  {
        graphics.Clear ();
        var touches =
Touch.GetData(0);
        if(touches.Count > 0 &&
touches[0].Status ==
TouchStatus.Down)
          done = true;
        UISystem.Update (touches);
```

```
                UISystem.Render ();

                graphics.SwapBuffers();
            }
        }
    }
}
```

Press *F5* to run the code and you should see the following:



This is the iconic "Hello World" application; the text **Hello world** will be displayed, centered on your screen.

How it works...

This example doesn't make use of the GameEngine2D framework, so we start off creating a new `GraphicsContext`. Next we initialize the `UISystem` singleton (similar in function to the `Director` singleton) passing in our newly created graphics context (we will see how this process works with a GameEngine2D scene in an upcoming recipe). Next we create a `Scene` and a `Panel`, which will contain our label control. We then set the panel's dimensions to match the full extents of the screen.

Next we create a `Label` object, which will display our string on screen. We set the label to be displayed in the center by setting its `HorizontalAlignment` and `VerticalAlignment` to `Center` and `Middle`, respectively. This causes the text to be centered within the `Label` control itself, but we still need to center the label to the panel, which we do using `SetPosition`. Finally, we set our label's text with the `Text` property.

Now we add the label to the panel by passing it as a parameter to `AddChildLast`. Next we add the panel itself to the UI `Scene` by calling `AddChild` on scene's `RootWidget`, making the panel the root-level widget in the tree of UI controls. We will look at this hierarchy of controls in greater detail later. We then set our scene as the active `Scene` by calling `UISystem.SetScene`.

Finally, before our game loop begins, we configure our viewport to the same dimensions as the screen and set the clear color to black.

In our game loop, we start by clearing the screen with a call to `Clear`. We then get the current touch state, and if a touch has occurred we set our done flag, telling the application to exit. Next we update `UISystem` with the current touch data by calling `Update`, and then display the UI with a call to `Render`. Finally, we draw the updated screen with a call to `graphics.SwapBuffers()`.

There's more...

It is important to realize that a UI `Scene` class has nothing in common with the Physics and GameEngine2D `Scene` classes and cannot be interchanged in any way. They all simply share a somewhat common design and naming convention, but that is where the similarities end.

In addition to passing touch data in to `UISystem` when you call the `Update` method, you can also add gamepad support. In this case, you also pass in the gamepad data, as follows:

```
UISystem.Update(Touch.GetData(0),GamePad.GetData(
```

You are required to pass in the touch information when calling UISystem's `Update`, but gamepad data is completely optional. When interacting with gamepad, pressing any button will set focus on the control; then pressing a button again will be effectively the same as touching it.

> **Note**
>
> Just like the Physics and GameEngine2D libraries, the UI library is provided with full source code. If you installed everything to default locations, the full source code should be available at `C:\Program Files (x86)\SCE\PSM\source\UI`.

There is one very important point you should be aware of. All of the classes in the UI library are *not* thread safe. All members of `UISystem` and all created widgets should be created in the main thread!

See also

- See the *Creating a simple game loop* recipe in Chapter 1, *Getting Started*, for more information on `GraphicsContext` and the basics of the game loop
- See the *Creating and using hierarchies of widgets* recipe for a list of available widget controls

Using the UI library within a GameEngine2D application

In this recipe we will look at integrating the UI library with a GameEngine2D project. In the process we will demonstrate adding and handling a `Button` widget.

Getting ready

Load up **PlayStation Mobile Studio** and create a new project. Add a reference to `Sce.PlayStation.HighLevel.UI` and `Sce.PlayStation.HighLevel.GameEngine2D`. Additionally, we will be drawing a 960 x 544 (the Vita's native resolution) `SpriteUV` using the image `beach.png`; of course you can substitute your own. The entire project including images is available in `Ch6_Example2`.

How to do it...

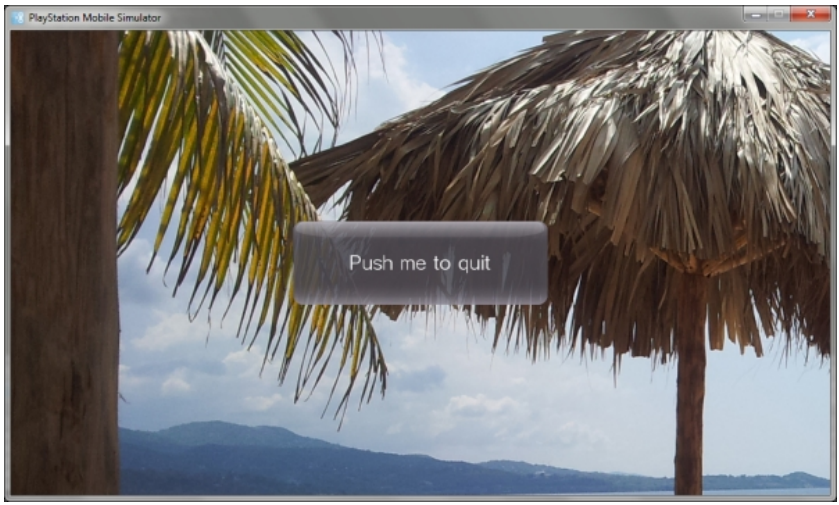
Open `AppMain.cs` and change the existing code to the following code:

```
using System;
using System.Collections.Generic;

using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Environment;
using Sce.PlayStation.Core.Graphics;
using Sce.PlayStation.Core.Input;
using
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Base;
using Sce.PlayStation.HighLevel.UI;
```

```
namespace Ch6_Example2
{
  public class AppMain {

    public static void Main (string[]
args)  {
      Director.Initialize();

UISystem.Initialize(Director.Instance.GL.Context)

      var scene = new
Sce.PlayStation.HighLevel.GameEngine2D.Scene();

scene.Camera.SetViewFromViewport();
      Texture2D texture = new
Texture2D("/Application/
beach.png",false);
      TextureInfo ti = new
TextureInfo(texture);
      SpriteUV background = new
SpriteUV(ti);
      background.Scale =
ti.TextureSizef;
      scene.AddChild(background);


      var uiScene = new
Sce.PlayStation.HighLevel.UI.Scene();
      Panel panel = new Panel();
      panel.Width =
Director.Instance.GL.Context.GetViewport().Width;
      panel.Height =
Director.Instance.GL.Context.GetViewport().Height

      bool quit = false;

      Button button = new Button();
      button.Name = "button1";
      button.Text = "Push me to
quit";
```

```
            button.Width = 300;
            button.Height = 100;
            button.PivotType =
PivotType.MiddleCenter;

button.SetPosition(panel.Width/2,panel.Height/2);
            button.TouchEventReceived +=
(sender, e) => {
              quit = true;
            };

            panel.AddChildLast(button);

uiScene.RootWidget.AddChildFirst(panel);
            UISystem.SetScene(uiScene);

Director.Instance.RunWithScene(scene,true);


            while(!quit){
              Director.Instance.Update();
              Director.Instance.Render();


UISystem.Update(Touch.GetData(0));
              UISystem.Render();


Director.Instance.GL.Context.SwapBuffers();
              Director.Instance.PostSwap();
            }

            texture.Dispose();
            ti.Dispose();
            Director.Terminate();
            UISystem.Terminate();
        }
    }
}
```

Hit *F5* to run the application and you should see the following screenshot:



In addition to a background of a beautiful Jamaican beach, a button labeled **Push me to quit** will be centered on the screen. Predictably enough, if you click on the button, the application will quit.

How it works...

This recipe starts off initializing both the GameEngine2D `Director` singleton, as well as the `UISystem` singleton. `UISystem` is initialized by passing in the GL context from within the `Director` singleton. Then it creates a (GameEngine2D) `Scene`, configures the camera, creates a `Texture2D` and a `TextureInfo`, which are used to create a `SpriteUV` object, which is scaled and added to the scene. If you have read the last couple of chapters already, this code should all be intimately familiar at this point.

Next we create a UI `Scene`, aptly named `uiScene`. Again, this `Scene` class has absolutely nothing in common with other `Scene` classes and cannot be interchanged; in fact, you may need to fully resolve the namespace to distinguish between the two classes. We then create a `Panel` object to hold our controls and size it to the full dimensions of the device. We also declare a `bool` object named `quit` to control program execution.

Then we declare a new `Button` widget. We set its name (ID) to `button1` and its displayed `Text` to `Push me to quit`. Next we set the button width and height and set the pivot point to the middle of the button using `PivotType`. `PivotType` is similar to GameEngine2D Node's `Pivot`, except instead of taking the x and y coordinates, it takes a `PivotType enum`. Next we set the button position to the center of the panel and then register a `TouchEventReceived` handler, which is a simple function that sets the `quit` flag to `true`.

Next we add the button to the panel using `AddChildLast` and then add the panel to `uiScene` by adding it to `RootWidget`, this time using `AddChildFirst` (when there is only one widget, there is no functional difference between `AddChildLast` and `AddChildFirst`). Finally, we set `uiScene` as the active `Scene` by calling `UISystem.SetScene()` and start the (GameEngine2D) scene running with a call to `Director.RunWithScene()`, passing in our scene and false, indicating we will manually handle the event loop.

In the event loop we make the mandatory `Update`, `Render`, `SwapBuffers`, and `PostSwap` calls. Additionally, after the `Director` singleton has been updated and drawn, we call `UISystem.Update()` and `UISystem.Render()`. After our event loop, we do some cleanup, including calling `Terminate` on both of our singletons.

There's more...

It is critically important that `UISystem.Render()` is called after `Director.Instance.Render()`. `UISystem` uses the GL context you passed in when you called `UISystem.Initialize()` to perform all of its drawing. This context is shared between the `Director` and `UISystem` objects and if you do not call `UISystem` object's `Render` after `Director` object's `Render`, the UI will be overdrawn by the GameEngine2D render process!

See also

- See Chapter 3, *Graphics with GameEngine2D*, for more

details on drawing graphics using GameEngine2D

Creating and using hierarchies of widgets

This recipe looks at the relationship between widgets. We are going to build a scrollable panel that contains another panel that is filled with an array of buttons and then demonstrate how they respond as a single unit.

Getting ready

Load up **PlayStation Mobile Studio** and create a new project. Add a reference to `Sce.PlayStation.HighLevel.UI`. The project is available in `Ch6_Example3`.

How to do it...

Open `AppMain.cs` and edit the code to match the following:

```
using System;
using Sce.PlayStation.Core;
using Sce.PlayStation.Core.Graphics;
using Sce.PlayStation.Core.Input;
using Sce.PlayStation.HighLevel.UI;

namespace Ch6_Example3
{
  public class AppMain {
    public static void Main() {
      GraphicsContext graphics = new
GraphicsContext();
      UISystem.Initialize(graphics);

      var scene = new
Sce.PlayStation.HighLevel.UI.Scene();
      var scrollPanel = new
Sce.PlayStation.HighLevel.UI.ScrollPanel();
```

```
      scrollPanel.SetPosition(0,0);
      scrollPanel.Width =
graphics.Screen.Width/2;
      scrollPanel.Height =
graphics.Screen.Height;
      scrollPanel.VerticalScroll =
true;
      scrollPanel.HorizontalScroll =
false;

scrollPanel.ScrollBarVisibility=
ScrollBarVisibility.ScrollableVisible;
      scrollPanel.PanelWidth =
graphics.Screen.Width/2;
      scrollPanel.PanelHeight =
100*10;

      var panel = new
Sce.PlayStation.HighLevel.UI.Panel();
      panel.Width =
graphics.Screen.Width/2;
      panel.Height = 100*10;
      panel.Clip = true;

      for(int i = 0; i < 10; i++)  {
        Button button = new Button()
        { X=0,Y=100*i, Width = 300,
Height=100, Name = "button"+i,
Text="Button #"+i };

        button.ButtonAction +=
(sender, e) => {
          if(scrollPanel.X ==0)
scrollPanel.X =
graphics.Screen.Width/2;
          else scrollPanel.X = 0;
        };
        panel.AddChildLast(button);
      }
```

```
scrollPanel.AddChildFirst(panel);

scene.RootWidget.AddChildLast(scrollPanel);

      UISystem.SetScene(scene);

graphics.SetViewport(0,0,graphics.Screen.Width,gr
      graphics.SetClearColor(new
Vector4(0,0,0,255));

      while(true)  {
        graphics.Clear();


UISystem.Update(Touch.GetData(0));
        UISystem.Render();

        graphics.SwapBuffers();
      }
    }
  }
}
```

Press the *F5* key to run your application:

An array of 10 buttons will be drawn on the left-hand side of the screen. Touch and drag to scroll the screen. Tapping a button will cause the entire collection of buttons, the containing panel, and the scroll bars all to move to the other side of the screen. Stop the application using the menu **Run** | **Stop** or by hitting *Shift* + *F5* in **PlayStation Mobile Studio**.

How it works...

This recipe also doesn't make use of the GameEngine2D library, so we have to create a `GraphicsContext`, then initialize the `UISystem` singleton. Next we create a UI `Scene` object.

The first widget we create is `ScrollPanel`, which is a special panel that scrolls its contents if they exceed the size of the panel. We set its width to be half the screen size and the height to be full screen, and enable vertical scrolling but not horizontal. We also set the scroll bars to be visible by setting `ScrollBarVisibility` to `ScrollBarVisibility.ScrollableVisible`. We then set `PanelWidth` to half the screen size and set `PanelHeight` to `1000`, large enough to accommodate 10 100-pixel high buttons. `PanelHeight`/`PanelWidth` represents the dimensions of the panel `ScrollPanel` contains.

Speaking of which, we create the inner panel next, which is a

standard `Panel` object. It predictably enough uses the same dimensions as the ones we just used for `PanelHeight` and `PanelWidth`. We set `Clip` to `true`, clipping the nonvisible portions of the widget.

Next we loop 10 times, creating 10 separate `Button` widgets, all with the same values except the y position, which we increment by 100 pixels with each iteration of the loop. For each button we define the same lambda function for `ButtonAction`. This function simply moves `scrollPanel` between the left- and right-hand side of the application each time a button is pressed. Finally at the end of each loop iteration the newly created `Button` is added to the panel.

We then add panel to `scrollPanel` by calling `AddChildFirst`, then add `scrollPanel` to the scene's `RootWidget`. With all the controls created and displayed we set our UI scene active with a call to `SetScene`, configure the viewport and clear color, and then loop forever, clearing the screen, updating and rendering the UI, and displaying it to the user.

There's more...

The position of a child widget is relative to the widget that contains it. All UI coordinates are relative to the upper-left corner of the screen for `RootWidget`, and the upper-left corner of the container for child widgets. Therefore the origin (0, 0) is at the upper-left corner of the screen, instead of the lower-left as used in GameEngine2D.

There are two kinds of controls in UI library, common widgets and container widgets. Container widgets are widgets that can contain other widgets, such as `Panel` and `Dialog`. Common widgets are your various UI components including `Button` and `Label`. A container widget is still a widget and has all of the same abilities, but the reverse is not true.

The following sections show the available UI controls in the PlayStation Mobile UI library.

Container widgets

The following are the container widgets:

- Dialog
    - MessageDialog
- GridListPanel
- ListPanel
- LiveScrollPanel
- Panel
    - ListPanelItem
    - LiveJumpPanel
    - LiveSpringPanel
    - UIAnimationPlayer
- RootWidget
- ScrollPanel

Common widgets

The following are the common widgets:

- AnimationImageBox
- BusyIndicator
- Button
- CheckBox
- ContainerWidget
- DatePicker
- EditableText
- ImageBox
- Label
- LiveFlipPanel
- LiveListPanel
- LiveSphere
- PagePanel
- PopupList
- ProgressBar
- ScrollBar
- Slider
- TimePicker

`RootWidget` is a special widget that represents the root of the widget tree. Each UI scene has `RootWidget` and it forms the base of all widget hierarchies. Beyond the one defined in scene, `RootWidget` classes are never declared. `ContainerWidget` is the parent class of all container widgets and is not used directly.

Creating a UI visually using UIComposer

This recipe is going to use the UIComposer application to create a simple user interface.

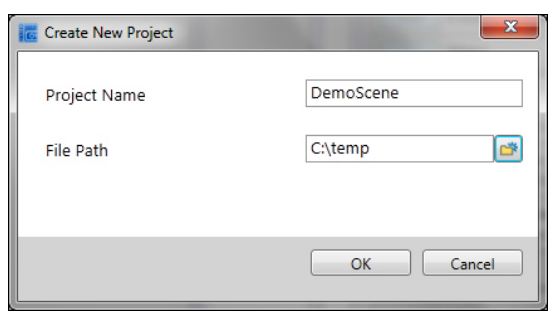
Getting ready

Load up **UIComposer**; there should be a **Start Menu** entry in the `PlayStation Mobile` folder. If you installed with default settings, it should also be available at `C:\Program Files (x86)\SCE\PSM\tools\UIComposer` if your OS is 64 bit, or `C:\Program Files\SCE\PSM\tools\UIComposer` if your OS is 32 bit.

How to do it...

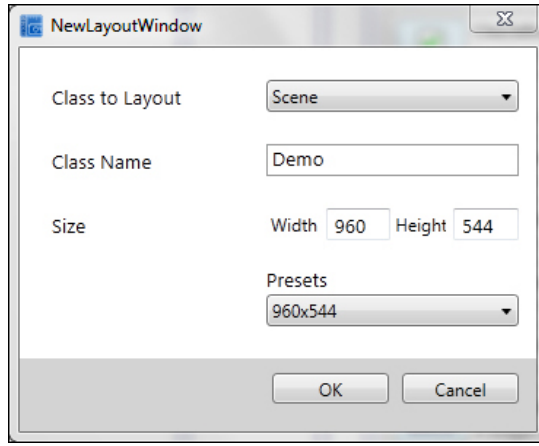
To create a UI visually using UIComposer perform the following steps:

1. Select **File** | **Create New Project**.
2. In the **Create New Project** dialog box, fill in the **Project Name** and **File Path** fields with whatever values you like (remember the location). Click on **OK** when finished:

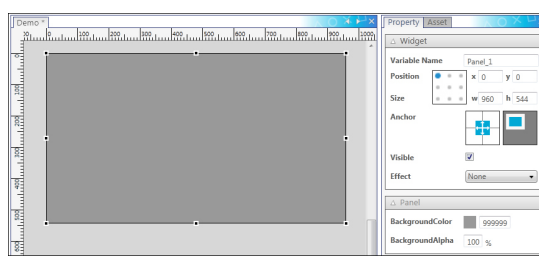


3. In the **NewLayoutWindow** dialog box, set **Class to Layout** to **Scene**, **Class Name** to `Demo`, and for **Size** select **960x544** (the PlayStation Vita's resolution), then

click on **OK**:

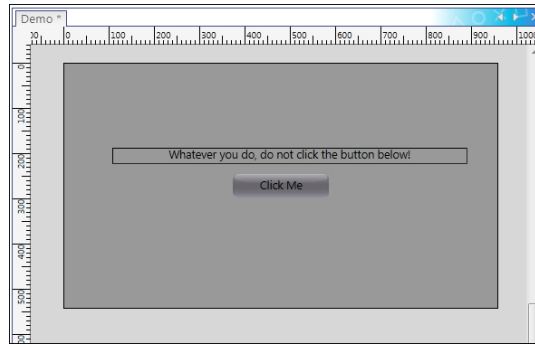


4. You should now have an editable surface you can drag, where you can essentially draw the UI. From the `WidgetList` area, locate the **Panel Widgets** section and drag a `Panel` widget over to the design surface.
5. Now set the properties of the `Panel` widget using the **Property** tab. **Variable Name** is the actual C# name that this widget will be assigned. Drag and position the panel in design surface to take up the full screen, or set the dimensions from the **Property** tab. Set **x** and **y** to `0`, **w** to `960`, and **h** to `544`, as shown in the following screenshot:



6. Drag a `Button` widget to the to the design surface, then center it by clicking on the **Align Vertical Center In Container** and **Align Horizontal Center in Container** toolbar buttons (on the left-hand side of the screen). In the `Text` property, change it from `Button` to `Click`

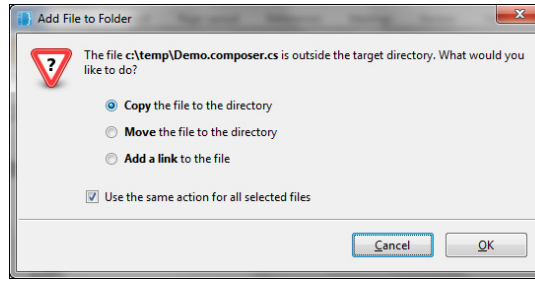
`Me`. Drag a `Label` widget above the button; you will notice that snapping lines will appear to help you position it relative to other controls. The end results should look something like the following screenshot:



7. Select the menu **File** | **Build**. A dialog should appear titled **Code Generating Progress** with a list of generated files. You can optionally save your project for editing it later with the menu **File** | **Save Project**.

   The files you just generated can now be used for importing into any existing project.

8. Open a solution in **PlayStation Mobile Studio** and be sure to add the UI library as a reference. Then import the files we just generated. Simply right-click on the project in the **Solution** panel and select **Add** | **Add Files...**, navigate to and select `Demo.cs`, `Demo.Composer.cs`, and `UIStringTable.cs` and click on **Open**. You will be prompted, as shown in the following screenshot:



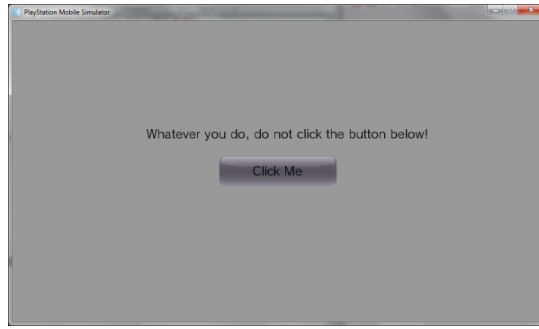
9. If you want to be able to make further changes to the

design in UIComposer, select **Add a Link to the file**. Otherwise you can select **Copy the file to the directory** or **Move the file to the directory**.

Now that these files have been included in your project, making use of this class is trivial. The following code loads the UI we just created:

```
    public static void Main
(string[] args)
    {
      var graphics = new
GraphicsContext ();
      DemoScene.Demo demo =
new DemoScene.Demo();


UISystem.Initialize(graphics);

UISystem.SetScene(demo);

      graphics.SetClearColor
(0.0f, 0.0f, 0.0f, 0.0f);
      while (true) {
        graphics.Clear ();

UISystem.Update(Touch.GetData(0));
        UISystem.Render();
        graphics.SwapBuffers
();
      }
    }
```

Then if you run this code, you will see the following screenshot:



The code doesn't actually do anything, but it does demonstrate the UI in action.

How it works...

UIComposer is essentially a code generator, writing the UI code for you. Thanks to the use of partial classes, you can implement logic in the `ClassName.cs` file, such as your `ButtonEvent` handlers, and leave generated code alone in the `ClassName.Composer.cs` file. This allows you to make future edits in UIComposer, without overwriting the customizations you make. It is still the programmer's responsibility to actually wire the UI up to… well, do stuff.

There's more...

In addition to creating complete `Scene` objects, UIComposer, it can also be used to create the `Panel`, `Dialog`, and `ListPanelItem` derived classes.

UIComposer has the ability to create layouts for both portrait and landscape modes. Use the menu item **View** | **Canvas Orientation** | **Landscape** and **View** | **Canvas Orientation** | **Portrait** to toggle between the two modes. The generated code will then handle initializing the widgets for the selected layout and provide a mechanism for switching between them.

See also

- See the *Creating and using hierarchies of widgets* recipe

for a complete list of available widgets

Displaying a MessageBox dialog

This short recipe is going to demonstrate displaying a message box dialog with an **OK** or **Cancel** prompt and show how to handle the user's response.

Getting ready

Load up **PlayStation Mobile Studio** and create a new project. Add a reference to `Sce.PlayStation.HighLevel.UI`. The entire project is available in `Ch6_Example4`. Copy the game's existing code from an earlier recipe in this chapter, as we are only going to cover the specifics of creating and interacting with the dialog in this recipe.

How to do it...

Add the following code during the update process to create and display `MessageDialog`:

```
var gamePadData = GamePad.GetData
(0);
if((gamePadData.Buttons &
GamePadButtons.Cross) ==
GamePadButtons.Cross)
{
  var dialog =
MessageDialog.CreateAndShow(MessageDialogStyle.Ok
Prompts",
                              "Are
you sure you wish to continue?");
  dialog.ButtonPressed += (sender, e)
=> {
    if(dialog.Result ==
DialogResult.Ok)
```

```
System.Diagnostics.Debug.WriteLine("Continuing");
    else

System.Diagnostics.Debug.WriteLine("The
user pressed cancel");
  };
}
```

In your main loop, make sure you update `UISystem` with the following code:

```
UISystem.Update (Touch.GetData(0));
UISystem.Render();
```

Now hit *F5* to run your application, then press the **X** button (or *S* in the simulator) and you should see the following screenshot:

Depending on which button you choose, a different message will be written to the debug console in Studio.

How it works...

We start off by retrieving the gamepad state, then check to see if the **X** button has been pressed. If it has, we create and display a message box using the static method `MessageDialog.CreateAndShow`, passing the value `MessageDialogStyle.OkCancel` indicating that we want the **OK** and **Cancel** buttons displayed. We also pass in the message box title and message text. Then we define a method to be called when the user clicks on a button, which checks the `Result` property to determine which button the user clicked. In either case, we print a message to the debug console using `System.Diagnostics.Debug.WriteLine()`, an easy way to output debug information.

There's more...

You can create a dialog class that is much more sophisticated than `MessageDialog` by deriving a class from `Dialog`. `Dialog` is one of the supported types generated by UIComposer, so you can use that tool to visually build more complex dialog boxes.

See also

- See the *Creating a UI visually using UIComposer* recipe for more details on using UIComposer, which can be used to create a custom dialog class

Handling touch gestures and using UI effects

In this recipe we are going to look at handling touch gestures such as double tapping and swiping. Additionally, in response to certain gestures, we are going to apply a couple of the built-in UI special effects.

Getting ready

Load up **PlayStation Mobile Studio** and create a new project. Add a reference to `Sce.PlayStation.HighLevel.UI`. We are going to be re-using our `FA-18h.png` image, but feel free to substitute any appropriate image. The entire project including images is available in `Ch6_Example4`.

How to do it...

Open `AppMain.cs` and replace `Main` with the following code:

```
public static void Main (string[]
args)  {
  var graphics = new GraphicsContext
();
  UISystem.Initialize(graphics);
  graphics.SetClearColor (0.0f, 0.0f,
0.0f, 0.0f);

  Panel panel = new Panel();
  panel.Width =
graphics.Screen.Width;
  panel.Height =
graphics.Screen.Height;

  ImageBox imageBox = new ImageBox();
  imageBox.Image = new ImageAsset("/
```

```
Application/FA-18h.png");
  imageBox.Width =
imageBox.Image.Width;
  imageBox.Height =
imageBox.Image.Height;
  imageBox.PivotType =
PivotType.MiddleCenter;

imageBox.SetPosition(panel.Width/2,panel.Height/2
  panel.AddChildLast(imageBox);
  FlickGestureDetector flickDetector
= new FlickGestureDetector();
  flickDetector.Direction =
FlickDirection.Horizontal;
  flickDetector.FlickDetected +=
delegate(object sender,
FlickEventArgs e) {

System.Diagnostics.Debug.WriteLine("Flick");
    if(e.Speed.X <0)
      new
MoveEffect(imageBox,2000.0f,0,imageBox.Y,MoveEffe
    else
      new
MoveEffect(imageBox,2000.0f,graphics.Screen.Width
();
  };

  DoubleTapGestureDetector
doubleTapDetector = new
DoubleTapGestureDetector();
  doubleTapDetector.DoubleTapDetected
+= delegate(object sender,
DoubleTapEventArgs e) {
    new
BunjeeJumpEffect(imageBox,1.0f).Start();
  };

  DragGestureDetector dragDetector =
new DragGestureDetector();
```

```
  dragDetector.DragDetected +=
delegate(object sender, DragEventArgs
e) {

imageBox.SetPosition(e.WorldPosition.X,e.WorldPos
  };


imageBox.AddGestureDetector(flickDetector);

imageBox.AddGestureDetector(doubleTapDetector);

imageBox.AddGestureDetector(dragDetector);

  var scene = new Scene();

scene.RootWidget.AddChildFirst(panel);
  UISystem.SetScene(scene);

  while(true) {
    graphics.Clear();

UISystem.Update(Touch.GetData(0));
    UISystem.Render();
    graphics.SwapBuffers();
  }
}
```

Hit *F5* to run your code and you should see the following screenshot:

The F18 sprite will be centered on the screen. Touch and drag to move the sprite around the screen, touch and "flick" left or right to throw the sprite to either side of the screen, or double tap it to apply a bungee jump style animation.

How it works...

This code starts off as usual, creating `GraphicsContext` and initializing `UISystem`. Next we create a full screen `Panel` to hold our widget, and then create an `ImageBox` widget. The image within `ImageBox` is `ImageAsset`, which we create by providing the image file location. We then set the `ImageBox` size to match the source image, and set its `PivotType` to `MiddleCenter`, making translations happen relative to the middle of `ImageBox`. Then we position the image in the middle of the panel and finally add it to the panel.

Now we declare a series of gesture detectors, starting with `FlickDetector`. We configure it to check for only horizontal movement and then register a delegate that will be called when a flick is detected. If a flick occurs, we check to see what the speed is. If it is moving in a negative x direction, this means it was flicked to the left, so we create `MoveEffect`. `MoveEffect` is targeted at `imageBox` and has a duration of 2000 milliseconds, a destination of the middle-left side of the screen and an elastic style interpolation

effect. In the event the user flicks right, we perform the same action, but instead target the middle-right side of the screen.

Next we register `DoubleTapGestureDetector` and create a delegate to be called when a double tap occurs. In the event of a double tap we perform `BungieJumpEffect` on `ImageBox`, with an elasticity of 1.0 (valid values are 0.0f to 1.0f), and immediately run the effect with a call to `Start`.

Then we register `DragGestureDetector`. Just like the others, it is handled with a delegate, which simply updates the `ImageBox` location with a call to `SetPosition` and passing in the drag location passed in as part of `DragEventArg`. The end result is that, as the touch location changes, `ImageBox` will be moved to that position.

We then have to wire all of the various gesture detectors to our widget. This is accomplished by calling the `ImageBox` widget's `AddGestureDetector` for each of the detectors we created. The remaining code is now the familiar infinite event loop.

There's more...

In addition to `DoubleTapGestureDetector`, `DragGestureDetector`, and `FlickGestureDetector`, there are also more detectors. `LongPressGestureDetector` is used to detect a tap and hold action. `PinchGestureDetector` is for detecting a two finger touch and drag, either inward or outward. Finally, there is `TapGestureDetector` for detecting a single tap gesture.

There are also a ton of built-in animation effects in addition to `MoveEffect` and `BunjeeEffect` . The following is a complete list of UI effect objects:

- `BunjeeJumpEffect`
- `DelayedExecutor`
- `FadeInEffect`
- `FadeOutEffect`
- `FlipBoardEffect`

- `JumpFlipEffect`
- `MoveEffect`
- `SlideInEffect`
- `SlideOutEffect`
- `TiltDropEffect`
- `UIMotion`
- `ZoomEffect`

See also

- See the *Loading, displaying, and translating a textured image* recipe from Chapter 1, *Getting Started* for details on how to add an image resource to your project and set its build action

Handling language localization

In this recipe we look at how to support multiple languages in your user interface using UIComposer.

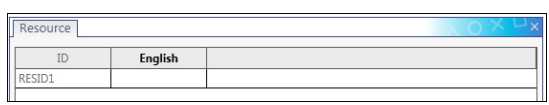
Getting ready

Load up **UIComposer**, either load the UI project we made earlier or create a new one. Be sure to add at least one widget with a `Text` property, such as a `Label` widget.

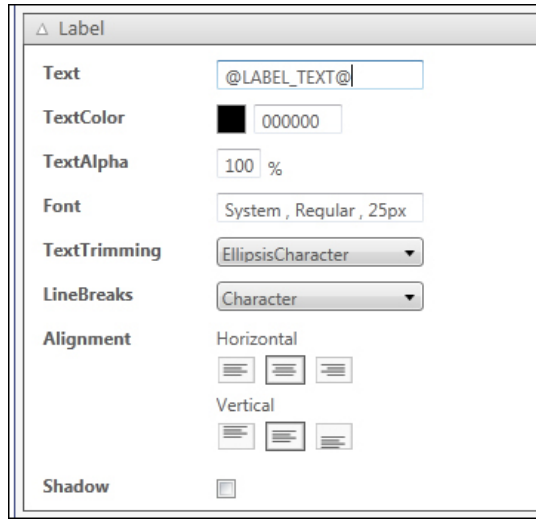
How to do it...

To handle language localization perform the following steps:

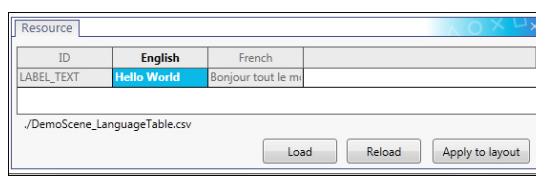
1. In **UIComposer**; select the menu **Window** | **Language Table Window** to enable the text editing functionality. You may need to click on the **Reload** button for the table to render properly. It should look like the following screenshot:



2. Right-click on the empty space to the right of **English** and select **Add Column**; a dialog box will appear. Enter the language name you want to support; in this example I am going to use French. Then click on **OK** and the column will be added.
3. In the designer, select your label. In the dialog area, click on a `Label` widget. In the **Property** window, set **Text** to **@LABEL_TEXT@**, as shown in the following screenshot:



4. Then back in the **Language Table** window add an entry to **LABEL_TEXT**; in the **English** column enter `Hello World` and in the **French** column enter `Bonjour tout le monde`. Then click on the **Apply to layout** button:



The design surface should now be updated to show your modified text. You can toggle between active languages by clicking on the language column in the **Language Table** window.

5. Build your UI project using either the menu **File** | **Build** or by hitting *F5*. In addition to the two files representing your UI, a third file `UIStringTable.cs` will be generated, such as the following one:

```
namespace DemoScene
{
    public static class
UIStringTable
    {
```

```
        static string[]
currentTable;
        static UILanguage
currentLanguageId;
        static UILanguage
defaultLanguageId;
        static string[][]
textTable;

        public static string
Get(UIStringID id)
        {
            return
currentTable[(int)id];
        }

        public static
UILanguage UILanguage
        {
            get { return
currentLanguageId; }
            set
            {
                if
(currentLanguageId != value)
                {

currentLanguageId = value;

currentTable =
textTable[(int)currentLanguageId];
                }
            }
        }

        static
UIStringTable()
        {
            textTable = new
string[][]
```

```
                {
                    new string[]
                    {
                        "Hello
World",
                    },
                    new string[]
                    {
                        "Bonjour
tout le monde",
                    },
                };

defaultLanguageId =
UILanguage.English;

currentLanguageId =
defaultLanguageId;
            currentTable =
textTable[(int)currentLanguageId];
        }
    }

    public enum UIStringID :
int
    {
        LABEL_TEXT = 0,
    }

    public enum UILanguage :
int
    {
        English = 0,
        French = 1,
    }
}
```

The generated UI classes will automatically make use of this class. You can then change languages by setting the

`currentLanguageId` property in your applications code.

There's more...

You can also save the language table to import from a **CSV** (**comma separated values**) file. This enables you to create and maintain your translations in a program other than UIComposer, such as Microsoft Excel. Strings can be any valid Unicode value.

See also

- See the *Creating a UI visually using UIComposer* recipe for more information on using UIComposer

Chapter 7. Into the Third Dimension

In this chapter, we will cover the following:

- Creating a simple 3D scene
- Displaying a texture 3D object
- Implementing a simple camera system
- A fragment (pixel) shader in action
- A vertex shader in action
- Adding lighting to your scene
- Using an offscreen frame buffer to take a screenshot

Introduction

In this chapter, we are going to learn about creating 3D graphics using the `Sce.PlayStation.Core.Graphics` library. The graphics library included with PlayStation Mobile is a fairly thin layer on top of OpenGL ES, so if you have done some prior OpenGL programming, this chapter should be immediately comfortable. Additionally, PlayStation Mobile makes heavy use of the Cg shader programming language, which we will also be covering.

Creating a simple 3D scene

In this recipe we are going to create about the simplest 3D scene possible, a single red triangle centered on the screen.

Getting ready

Load **PlayStation Mobile Studio** and create a new project. You can download the complete project as `Ch7_Example1`

How to do it...

1. Change `AppMain.cs` to match the following code:

```
using System;
using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.Core.Input;

namespace Ch7_Example1
{
public class AppMain {
public static void Main
(string[] args){
var graphics = new
GraphicsContext ();
graphics.SetClearColor
(255.0f, 255.0f, 255.0f,
255.0f);

var projectionMatrix =
Matrix4.Perspective(FMath.Radians(45.0f),

var viewMatrix =
Matrix4.LookAt(new
Vector3(0.0f,1.0f,-3.0f),
```

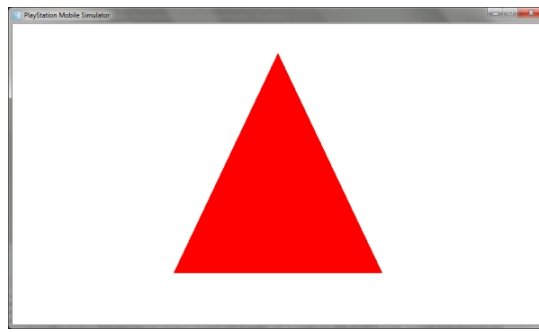
```
   new
Vector3(0.0f,0.0f,0.0f),

   Vector3.UnitY);

var shaderProgram = new
ShaderProgram("/Application/
shaders/simple.cgx");
var vertexBuffer = new
VertexBuffer(3,VertexFormat.Float3);

shaderProgram.SetUniformBinding(0,"WorldV
shaderProgram.SetAttributeBinding(0,"a_Po
float[] positions = {
            0.0f, 1.0f,
0.0f,
            -1.0f, -1.0f,
0.0f,
            1.0f, -1.0f,
0.0f,
        };

vertexBuffer.SetVertices(0,positions);

bool done = false;
while(!done) {
graphics.Clear ();

if((GamePad.GetData(0).Buttons
& GamePadButtons.Cross) ==
GamePadButtons.Cross)
done = true;

var viewProjection =
projectionMatrix *
viewMatrix;
shaderProgram.SetUniformValue(0,
ref viewProjection);
graphics.SetShaderProgram(shaderProgram);
```

```
graphics.SetVertexBuffer(0,vertexBuffer);
graphics.DrawArrays(DrawMode.TriangleStri

graphics.SwapBuffers ();
}

graphics.Dispose();
shaderProgram.Dispose();
vertexBuffer.Dispose();
}
}
}
```

2. Next expand the `shaders` folder, open `Simple.fcg`, and edit it as follows:

```
void main( float4 out Color
: COLOR,uniform float4
MaterialColor )
{
Color = float4(1,0,0,1);
}
```

3. Press *F5* to run the code and you should see the output, as shown in the following screenshot:



A single red triangle will be displayed in the center of the screen. You can press the *X* button or the *S* key to exit.

How it works...

If you have already read Chapter 1, *Getting Started*; this recipe might look virtually identical to the Hello World application, as the process is basically the same. In fact it is a bit simpler.

We start off creating the `GraphicsContext`, which is an encapsulation of the underlying OpenGL context and is the heart of all graphic related activity. We then set the clear color to solid white. Next we create a pair of `Matrix4` objects, the `projectionMatrix` and the `viewMatrix`. The `projectionMatrix` is responsible for converting the world from 3D view space to clip space, or in plainer less accurate language, to your screen. When creating the projection matrix you pass in the field of view (in radians), the screen's aspect ratio plus the near and far clip planes of the view frustrum.

The `viewMatrix`, on the other hand, represents your virtual eye within the 3D world. When creating the `viewMatrix` you pass in a trio of `Vector3` objects, one representing the eye's location in 3D space, the next representing where the eye is looking. The third vector indicates where "Up" is; in this case we set the Y axis as the Up vector. The world (not used here), view, and projection matrices work together to transform the 3D scene to your devices screen. We cover this process in more detail later in this chapter.

Next we load the `simple.cgx` shader file, which PlayStation Mobile Studio automatically created for us. You may notice the file extension is different from the files in your script folder; don't worry, we will cover why this is shortly. Next we create a `VertexBuffer`, three items in length, containing `Float3` objects. Then we bind to the shader values `WorldViewProj` and `a_Position` using the methods `SetUniformBinding` and `SetAttributeBinding`.Think of binding simply as setting a relationship between an application variable and a shader variable. We will cover both methods in more detail soon.

Now that our shader is loaded and properly bound, we need to create our triangle. We do this manually in an array of floats named `positions`, each value represents the x, y, or z coordinate making up each individual vertex. Therefore the first vertex of our triangle is (0.0f, 1.0f, 0.0f), the second is (-1.0f, -1.0f, 0.0f), and so on. We then load our vertices into the `vertexBuffer` by calling

`SetVertices()`, the 0 representing the index of the buffer we want to load. As you will see later, you can load multiple buffers into a single `VertexBuffer`, in this case, we only have the one.

Next is the event loop. Each pass through the loop we clear the screen, then check to see if the user has pressed the **X** button and, if they have, toggle the done variable to `true`, which will end our loop the next pass through. Then we calculate the `viewProjection` matrix; this is a combination of the View and Projection matrixes and is calculated by simply multiplying them together. We then pass this calculated matrix in to our shader at the 0th index that we bound earlier to the shader variable "`WorldViewProj`".

Finally, we load our shader into the graphics context with a call to `SetShaderProgram()` load our `VertexBuffer` by calling `SetVertexBuffer()`, which again can accommodate multiple buffers; so we set it to the 0th position since we have only one. Next we draw the actual triangle to the screen with a call to `DrawArrays`, telling it to render the `VertexBuffer` as a triangle strip, starting at `0` and with a count of `3`. Finally we display our rendered triangle onscreen by calling `SwapBuffers()`.

In addition to the graphics object, `ShaderProgram` and `VertexBuffer` are both `IDisposable`, so to prevent leaking memory, we `Dispose()` of each before exiting.

There's more...

`TriangleStrip` is just one of the primitives that you can render; in addition to `TriangleStrip`, you can render the following primitive types with `DrawArray()`:

- `Points`
- `Lines`
- `LineStrip`
- `Triangles`
- `TriangleFan`

`TriangleStrip` is the most commonly used type, as devices are

generally optimized to render in triangle strips. A `TriangleFan` is similar to a `TriangleStrip`, except every triangle in the fan shares a single vertex.

The `DrawArray` call we made earlier is a very expensive operation in terms of overhead. Care should be taken to batch as many calls together in a single `DrawArray` call as possible. You can batch together graphics operations that have a great deal in common (same shader and rendering settings).

See also

- See the *Loading, displaying and translating a textured image* recipe in Chapter 1, *Getting Started*, for another recipe demonstrating creating and rendering a simple 3D scene.

Displaying a textured 3D object

In this recipe we are going to implement the "Hello World" equivalent of the 3D graphics world. We will be creating a fully textured 3D crate, the most famous textured cube known to mankind.

Getting ready

Load **PlayStation Mobile Studio** and create a new project. Additionally, you will need a create texture map. I downloaded this texture from the site www.OpenGameArt.org. You can download this particular texture at http://bit.ly/TuKCo. You can download the complete project including images as `Ch7_Example2`.

How to do it...

1. Open `AppMain.cs` and replace its contents with the following code:

```
using System;
using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Graphics;
using
Sce.PlayStation.Core.Input;

namespace Ch7_Example2
{
public class AppMain {
public static VertexBuffer
MakeCube() {
    VertexBuffer
vertexBuffer = new
VertexBuffer(24, 36,
VertexFormat.Float3,
VertexFormat.Float2);

        float[] positions =
```

```
{
            -0.5f,  0.5f,
-0.5f, -0.5f,  0.5f,  0.5f,
0.5f,  0.5f, -0.5f, 0.5f,
0.5f,  0.5f, //top
             0.5f, -0.5f,
-0.5f, 0.5f, -0.5f,  0.5f,
-0.5f, -0.5f, -0.5f, -0.5f,
-0.5f,  0.5f,//bottom
            -0.5f,  0.5f,
0.5f, -0.5f, -0.5f,  0.5f,
0.5f,  0.5f,  0.5f, 0.5f,
-0.5f,  0.5f,//front
             0.5f,  0.5f,
-0.5f, 0.5f, -0.5f, -0.5f,
-0.5f,  0.5f, -0.5f, -0.5f,
-0.5f, -0.5f,//back
             0.5f,  0.5f,
0.5f, 0.5f, -0.5f,  0.5f,
0.5f,  0.5f, -0.5f, 0.5f,
-0.5f, -0.5f,//right
            -0.5f,  0.5f,
-0.5f, -0.5f, -0.5f, -0.5f,
-0.5f,  0.5f,  0.5f, -0.5f,
-0.5f,  0.5f//left
        };
        float[] texcoords =
{
            0.0f, 1.0f,
0.0f, 0.0f, 1.0f, 1.0f,
1.0f, 0.0f, //top
            0.0f, 1.0f,
0.0f, 0.0f, 1.0f, 1.0f,
1.0f, 0.0f, //bottom
            0.0f, 1.0f,
0.0f, 0.0f, 1.0f, 1.0f,
1.0f, 0.0f, //front
            0.0f, 1.0f,
0.0f, 0.0f, 1.0f, 1.0f,
1.0f, 0.0f, //back
```

```
            0.0f, 1.0f,
0.0f, 0.0f, 1.0f, 1.0f,
1.0f, 0.0f, //right
            0.0f, 1.0f,
0.0f, 0.0f, 1.0f, 1.0f,
1.0f, 0.0f //left
        };
        ushort[] indices = {
             0,  1,  2, 1,
3,  2,  //top
             4,  5,  6, 5,
7,  6,  //bottom
             8,  9, 10, 9,
11, 10,  //front
            12, 13, 14, 13,
15, 14, //back
            16, 17, 18, 17,
19, 18, //right
            20, 21, 22, 21,
23, 22  //left
        };

vertexBuffer.SetVertices(0,
positions);

vertexBuffer.SetVertices(1,
texcoords);

vertexBuffer.SetIndices(indices);

return vertexBuffer;
}

public static void Main
(string[] args){
var graphics = new
GraphicsContext ();
graphics.SetClearColor
(255.0f, 255.0f, 255.0f,
0.0f);
```

```
var projectionMatrix =
Matrix4.Perspective(FMath.Radians(45.0f),

var viewMatrix =
Matrix4.LookAt(new
Vector3(0.0f,1.0f,-3.0f),

   new
Vector3(0.0f,0.0f,0.0f),

   Vector3.UnitY);
var vertexBuffer =
MakeCube();
var texture = new
Texture2D("/Application/
crate.png",false);

var textureShader = new
ShaderProgram("/Application/
shaders/Texture.cgx");

textureShader.SetUniformBinding(0,
"WorldViewProj");

textureShader.SetAttributeBinding(0,
"a_Position");

textureShader.SetAttributeBinding(1,
"a_TexCoord");

bool done = false;
while(!done) {
if((GamePad.GetData(0).Buttons
& GamePadButtons.Cross) ==
GamePadButtons.Cross)
done = true;

graphics.Clear ();
var viewProjection =
projectionMatrix *
```

```
viewMatrix;
textureShader.SetUniformValue(0,
ref viewProjection);

graphics.SetShaderProgram(textureShader);
graphics.SetVertexBuffer(0,vertexBuffer);
graphics.SetTexture(0,texture);

graphics.Enable(EnableMode.CullFace,true)
graphics.DrawArrays(DrawMode.Triangles,0,

graphics.SwapBuffers ();
}
textureShader.Dispose();
texture.Dispose();
graphics.Dispose();
}
}
}
```

2. Next we need to create a new shader in the `shader` folder.
3. Right-click on the `shader` folder in the **Solution** window and select **Add** | **New File…**.
4. In the resulting dialog, on the left-hand side select **PlayStation Mobile**, then on the right-hand side select **Empty Shader File**. At the bottom in the **Name** field, call it `Texture`, then click on the **New** button.

   This will create a pair of files, `Texture.vcg` and `Texture.fcg`.

5. Open `Texture.vcg` and edit it to match the following code:

```
void main(float4 in
a_Position  : POSITION,
float2 in a_TexCoord  :
TEXCOORD0,
float4 out v_Position :
POSITION,
```

```
float2 out v_TexCoord :
TEXCOORD0,
uniform float4x4
WorldViewProj
   )
{
v_Position= mul( a_Position,
WorldViewProj );
v_TexCoord  = a_TexCoord;
}
```

6. Then open `Texture.fcg` and edit it to match the following code snippet:

```
void main(
float2 in  v_TexCoord     :
TEXCOORD0,
float4 out Color  : COLOR,
uniform sampler2D Texture0 :
TEXUNIT0)
{
Color = tex2D(Texture0,
v_TexCoord);
}
```

7. Now run the application and you should see the following output:

A fully textured 3D cube will appear centered on the screen. Press **X** or hit the *S* key to exit.

How it works...

We start off by defining a function named `MakeCube()`. We make it a function because we are going to be re-using this cube in future recipes. The first thing we do in the function is create a `VertexBuffer`. The first parameter for the `VertexBuffer` constructor is the number of vertices that will be in our buffer, and the second value is the number of indices. Next we pass in the format of each buffer we want to create; in our case we will be creating two. The first is of type `VertexFormat.Float3` and holds vertex information. The second is of type `VertexFormat.Float2` and holds 2D texture coordinates.

Now we populate the array of values that compose our cube. In our positions array, each value represents a coordinate composing our vertices. For example, the array starts with the values -0.5f, 0.5f, and -0.5f, which will result in the first vertex having an `x` value of `-0.5f`, a `y` value of `0.5f`, and a `z` value of `-0.5f`. We then continue to define all of the other vertices composing our cube. The first 12 values comprise the 4 vertices that in turn make up the triangles forming the top of our cube. All of these coordinates are relative to the origin. We then repeat the process for the bottom, front, back, right, and left faces.

Next we define our texture coordinates or UV values. In this case, the buffer is a `Vertex2` format, so the first value becomes (0,1), the second becomes (0,0), and so on. By setting the UV coordinates in order (clockwise or counterclockwise depending how the renderer is configured), we are telling OpenGL how to draw our texture on the face. By altering UV coordinates, you can rotate, skew, flip, and so on the texture on the face.

Finally we create our indices array. These values are offsets within the positions array and tell OpenGL how to draw our triangles. For example, the values (0,1,2) say that the triangle composing half of our top face is composed of the vertices at positions 0, 1, and 2 in the positions array. Therefore our first triangle is made up of the vertices (-0.5,0.5,-0.5),(-0.5,0.5,0.5), and (0.5,0.5,-0.5), while our second triangle is made using vertices 1,3, and 2, which are (-0.5,0.5,0.5), (0.5,0.5,0.5), and (0.5,0.5,-0.5). As you can see, these

two triangles share a pair of vertices and together form the quad that make up the top of our cube.

When we created the `VertexBuffer`, in the constructor we told it to expect two arrays. Therefore our `VertexBuffer` is now expecting a pair of arrays, one that will be made of Vector3 objects and one that will be made of Vector2 objects. We pass these in using the method `SetVertices()` in the same order they were defined in the constructor. Finally, we set the indices array with a call to `SetIndices()` and return our newly created `VertexBuffer`.

In our `Main` function, we start off creating our `GraphicsContext`, and then set the clear color to solid white. We set up our `projectionMatrix` to use a 45 degree field of view and an aspect ratio matching our screen's, with a nearly infinite viewing frustum. Then we create our `viewMatrix`, which is our location and view direction within the 3D scene. We position our viewing angle slightly up (Y axis) and back (Z axis) so that we will see our cube at a slight angle (otherwise it would look remarkably like a rectangle!). Since we are drawing our cube about the origin, we set that location as our view target.

Next we actually make our `VertexBuffer` by calling `MakeCube` and loading the `crate.png` into a `Texture2D` object. Then we load or `Texture.cgx` shader and bind the`WorldViewProj` script value and the `a_Position` and `a_TexCoord` attributes. Don't worry if this process is a bit confusing for now; we cover it in more detail in an upcoming recipe.

Now we start our event loop, looping until the user hits the *X* button or *S* key. With each pass through the loop, we clear the screen then create our `viewProjection` matrix by multiplying the projection and view matrixes together. We then load our shader, buffer, and texture in to the graphics context with calls to `SetShaderProgram`, `SetVertexBuffer`, and `SetTexture` respectively. We then enable culling, so that polygons further back in the scene will not draw over closer polygons (comment out this line to see the effect in action). Then we render the whole thing by calling `DrawArrays()`. Finally we display all our hard work on screen with a call to `SwapBuffers()`. After the event loop we

perform some cleanup before exiting.

There's more...

When texture mapping with PlayStation Mobile, to be compatible across devices, the following features are not available:

- Repeating non power of two textures
- Compressed textures
- Vertex textures
- Depth textures
- 3D textures
- Rendering to half float textures
- Rendering to luminance alpha textures

Graphics, textures, and shaders also have the following limits:

- MaxTextureSize 2048
- MaxCubeMapTextureSize 2048
- MaxRenderbufferSize 2048
- MaxVertexUniformVectors 128
- MaxFragmentUniformVectors 64
- MaxVertexAttribs 8
- MaxVaryingVectors 8
- MaxTextureImageUnits 8
- MaxAliasedLineWidth 8
- MaxAliasedPointSize 128

If these limitations seem confusing to you, do not worry, you generally will not run into most of them.

See also

- See the *A fragment pixel shader in action* and *A vertex shader in action* for more details on how shaders work and for more details on how a `cgx` file is generated

Implementing a simple camera system

In this recipe, we are going to create a simple camera that can be panned, rotated, and zomed in and out.

Getting ready

Load **PlayStation Mobile Studio** and create a new project. This recipe is going to build heavily on the code from the prior recipe, *Creating a 3D scene*. You may wish to duplicate the project; if you do not, be sure to copy `AppMain.cs`, `crate.png`, `texture.vcg`, and `texture.fcg` over to your new project. You can download the complete project as `Ch7_Example3`.

How to do it...

1. Add a new file, `Camera.cs` and enter the following code:

```
using System;
using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Input;
using
Sce.PlayStation.Core.Graphics;

namespace Ch7_Example3
{
public class Camera
{
private Matrix4 _projection;
private Matrix4 _view;
private Matrix4 _world;
private float _FOV;
private float _aspectRatio;
private Vector3 _position;
```

```
private Vector3 _target;

public Camera (float fov,
float aspectRatio, Vector3
position, Vector3 target)
{
_FOV = FMath.Radians(fov);
_aspectRatio = aspectRatio;
_position = position;
_target = target;
_world =
Matrix4.Translation(new
Vector3(0.0f,0.0f,0.0f));
}

public Matrix4 GetMatrix()
{
HandleInput();
_projection =
Matrix4.Perspective(_FOV,_aspectRatio,1.0
10000.0f);
_view =
Matrix4.LookAt(_position,_target,Vector3.

return _projection * _view *
_world;
}

public void
SetTarget(Vector3 target)
{
_target = target;
}

public void
SetPosition(Vector3
position)
{
_position = position;
}
```

```
private void HandleInput()
{
// Panning with dpad
if((GamePad.GetData(0).Buttons
& GamePadButtons.Up) ==
GamePadButtons.Up)
_world = _world *
Matrix4.Translation(new
Vector3(0.0f,-0.1f,0.0f));
if((GamePad.GetData(0).Buttons
& GamePadButtons.Down) ==
GamePadButtons.Down)
_world = _world *
Matrix4.Translation(new
Vector3(0.0f,0.1f,0.0f));
if((GamePad.GetData(0).Buttons
& GamePadButtons.Left) ==
GamePadButtons.Left)
_world = _world *
Matrix4.Translation(new
Vector3(-0.1f,0.0f,0.0f));
if((GamePad.GetData(0).Buttons
& GamePadButtons.Right) ==
GamePadButtons.Right)
_world = _world *
Matrix4.Translation(new
Vector3(0.1f,0.0f,0.0f));

// Zooming with shoulder
buttons ( Q/E )
if((GamePad.GetData(0).Buttons
& GamePadButtons.R) ==
GamePadButtons.R)
_world = _world *
Matrix4.Translation(new
Vector3(0.0f,0.0f,-0.1f));
if((GamePad.GetData(0).Buttons
& GamePadButtons.L) ==
GamePadButtons.L)
```

```
_world = _world *
Matrix4.Translation(new
Vector3(0.0f,0.0f,0.1f));
_view =
Matrix4.LookAt(_position,_target,Vector3.

// Rotate left right using
on screen touch.
var touch = Touch.GetData
(0);
if(touch.Count > 0)
{
if(touch[0].X < -0.1f)
_world = _world *
Matrix4.RotationY(FMath.Radians(1.0f));
if(touch[0].X > 0.1f)
_world = _world *
Matrix4.RotationY(-
FMath.Radians(1.0f));
}
}
}
}
```

2. Open `AppMain.cs` and edit the `Main` function to look similar to the following (changes highlighted):

```
public static void Main
(string[] args){
var graphics = new
GraphicsContext ();
graphics.SetClearColor
(255.0f, 255.0f, 255.0f,
0.0f);

Camera camera = new
Camera(45.0f,graphics.Screen.AspectRatio,
Vector3(0.0f,1.0f,-3.0f),new
Vector3(0.0f,0.0f,0.0f));
```

```
var vertexBuffer =
MakeCube();
var texture = new
Texture2D("/Application/
crate.png",false);
var textureShader = new
ShaderProgram("/Application/
shaders/Texture.cgx");

textureShader.SetUniformBinding(0,
"WorldViewProj");

textureShader.SetAttributeBinding(0,
"a_Position");

textureShader.SetAttributeBinding(1,
"a_TexCoord");
var done = false;
while(!done) {
if((GamePad.GetData(0).Buttons
& GamePadButtons.Cross) ==
GamePadButtons.Cross)
done = true;
graphics.Clear ();

var viewWorldProjection =
camera.GetMatrix();
textureShader.SetUniformValue(0,
ref viewWorldProjection);

graphics.SetShaderProgram(textureShader);
graphics.SetVertexBuffer(0,vertexBuffer);
graphics.SetTexture(0,texture);

graphics.Enable(EnableMode.CullFace,true)
graphics.DrawArrays(DrawMode.Triangles,0,

graphics.SwapBuffers ();
}
textureShader.Dispose();
```

```
vertexBuffer.Dispose();
graphics.Dispose();
}
```

3. Press *F5* to run your application and you should see output similar to the following screenshot:

Once again, a cube will be drawn centered on the screen. Now, however, you have control over the camera. You can zoom the camera in and out using the shoulder buttons (or *Q* and *E* keys), pan the camera using the d-pad, and rotate it by touching the left-hand side or right-hand side of the screen. Press the *X* button or the **S** key to exit.

How it works...

The bulk of our new changes take place in the new `Camera.cs` class. The class itself is relatively simple; we are creating a wrapper around the construction of the `WorldViewProjection` matrix, which is a combination of world, view, and projection matrixes. As mentioned earlier, the `View` matrix is your eye's position and target within the 3D world. The `Projection` matrix is what translates the world from 3D to clip space, or in other words, what is going to be displayed on your screen. The `World` matrix represents movement within the world. When you translate or rotate your position in the 3D world, it is done with the `World` matrix.

Our `Camera` class constructor takes the field of view, aspect ratio, initial eye position, and target. We also set up the `World` matrix

with a basically empty translation. `GetMatrix()` is the function that is called externally and is responsible for building each of the matrixes and combining them together. The actual logic used to construct the projection and view matrixes is exactly the same as prior recipes, just moved into an external class. Before building the resulting matrix, we call `HandleInput()` to see if the user has moved the camera.

`HandleInput` is fairly straightforward. If the user presses the d-pad, we move the camera in the corresponding direction. This is accomplished by multiplying the `World` matrix by a new `Translation` matrix in the appropriate direction along the X or Y axis. We then perform the same action if the user presses one of the shoulder buttons; this time instead we want to zoom the camera in and out. This is accomplished by creating a translation on the Z axis. Finally we check to see if the user tapped either the left-hand side or right-hand side of the screen, and if they have we rotate the screen accordingly. This is accomplished using `RotateY`, which creates a matrix that performs a rotation around the Y axis, and then multiply this matrix by our `World` matrix. The rotation is in radians, so we convert using the `FMath.Radians()` utility function. You may be wondering why I didn't make use of the analog sticks to control the camera. The simple answer is they aren't supported using the simulator so to make it easier to follow along for people that don't have a PS Vita device. I tended not to use them in the recipes.

Back in `AppMain.cs`, the only major changes is we no longer build each matrix by hand; instead we create and use the `Camera` object. Additionally, once each game loop, we get the updated matrix with a call to `GetMatrix()`, effectively updating the location in the world.

See also

- This recipe represents a relatively primitive camera class. www.OpenGL.org has some more detailed information on dealing with some of the more complex camera issues, such as framing a target. Although the examples are in C, the information translates almost perfectly. The link is http://bit.ly/U9EY0J.

A fragment (pixel) shader in action

In this recipe we create a simple fragment shader that will allow you to cycle the color on our textured cube.

Getting ready

Load **PlayStation Mobile Studio** and create a new project. Once again, this recipe's code will build off the code from the prior recipe, so simply copying that project is the easiest way to start. If you decide to create a new project, be sure to copy the `crate.png` texture over, as well as the code from `AppMain.cs`. The complete project is available as `Ch7_Example4`.

How to do it...

1. Add `AppMain.cs` and, assuming you copied the code from the prior recipe, change `Main` to match the following (changes are highlighted):

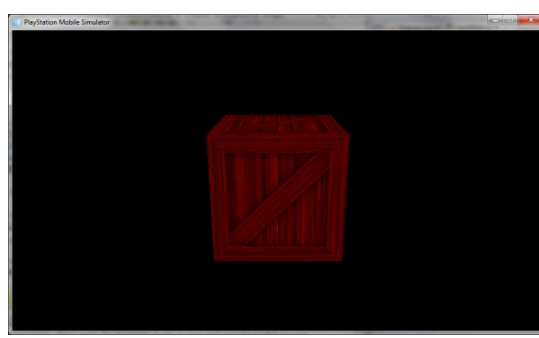
```
public static void Main
(string[] args){
var graphics = new
GraphicsContext ();
graphics.SetClearColor
(0.0f, 0.0f, 0.0f, 0.0f);

Camera camera = new
Camera(45.0f,graphics.Screen.AspectRatio,
Vector3(0.0f,1.0f,-3.0f),new
Vector3(0.0f,0.0f,0.0f));

var vertexBuffer =
MakeCube();
var texture = new
Texture2D("/Application/
```

```
crate.png",false);

var textureShader = new
ShaderProgram("/Application/
shaders/Texture.cgx");

textureShader.SetUniformBinding(0,
"WorldViewProj");


textureShader.SetAttributeBinding(0,
"a_Position");

textureShader.SetAttributeBinding(1,
"a_TexCoord");

Vector4 colorMask = new
Vector4(1,1,0,0);
float alpha = 1.0f;

bool done = false;
while(!done) {
graphics.Clear ();

var viewWorldProjection =
camera.GetMatrix();
textureShader.SetUniformValue(0,
ref viewWorldProjection);

if((GamePad.GetData(0).Buttons
& GamePadButtons.Cross) ==
GamePadButtons.Cross)
done = true;

if((GamePad.GetData(0).Buttons
& GamePadButtons.Triangle)
== GamePadButtons.Triangle){
colorMask = new
Vector4(0,1,1,0);
}
```

```
else
if((GamePad.GetData(0).Buttons
& GamePadButtons.Circle) ==
GamePadButtons.Circle){
colorMask = new
Vector4(1,0,1,0);
}
else
if((GamePad.GetData(0).Buttons
& GamePadButtons.Square) ==
GamePadButtons.Square){
colorMask = new
Vector4(1,1,0,0);
}
textureShader.SetUniformValue(textureShad
colorMask);

if((GamePad.GetData(0).Buttons
& GamePadButtons.Select) ==
GamePadButtons.Select){
if(alpha < 1.0f) alpha +=
0.01f;
}
if((GamePad.GetData(0).Buttons
& GamePadButtons.Start) ==
GamePadButtons.Start){
if(alpha > 0.0f) alpha -=
0.01f;
}
textureShader.SetUniformValue(textureShad

graphics.Enable(EnableMode.Blend);
graphics.SetBlendFunc(new
BlendFunc(BlendFuncMode.Add,BlendFuncFact

graphics.SetShaderProgram(textureShader);
graphics.SetVertexBuffer(0,vertexBuffer);
graphics.SetTexture(0,texture);

graphics.Enable(EnableMode.CullFace,true)
```

```
graphics.DrawArrays(DrawMode.Triangles,0,

graphics.SwapBuffers ();
}
vertexBuffer.Dispose();
textureShader.Dispose();
graphics.Dispose();
}
Now open Texture.fcg and
edit to match:
void main(
float2 in  v_TexCoord     :
TEXCOORD0,
float4 out Color  : COLOR,
uniform sampler2D Texture0 :
TEXUNIT0,
uniform float4 ColorMask,
uniform float Alpha)
{
float4 texelColor =
tex2D(Texture0, v_TexCoord);
Color = texelColor -
ColorMask;
Color.a = Alpha;
}
```

2. Now run the application and you will see the output similar to the following screenshot:



It is our trusty cube again, once again centered on the

screen, although you can now move the camera around using the same controls as the prior recipe. Now if you press the **Triangle**, **Circle**, or **Square** buttons, the color of the texture will change between red, blue, and green. Additionally, **Select** and **Start** will increase and decrease the alpha value, causing the cube to become more or less transparent. Once again, the *X* button will exit the application.

How it works...

The code in `AppMain.cs` is virtually unchanged from the prior recipe. The majority of the changes revolve around two new variables, `colorMask`, which is a `Vector4` representing the colors we *don't* want in our texture, and `alpha`, a float value representing the target alpha channel value of our texture. In our game loop, we update the `colorMask` and alpha values based on the user's actions. The following are the most important lines:

```
textureShader.SetUniformValue(textureShader.FindU
colorMask);
```

and

```
textureShader.SetUniformValue(textureShader.FindU
```

These two calls are what binds our C# based variables to a variable within the shader. The first call, for example, looks in the shader for a variable named `ColorMask` and sets its value to `ColorMask`.

If you look at `Texture.fcg` you will see in the `main` function declaration that we defined the parameters `uniformfloat4 ColorMask` and `uniform float Alpha`; these two variables get populated by these `SetUniformValue` calls.

The shader itself is really simple. `tex2D` is a Cg function that performs a texture lookup at a given position. Essentially it gives us

the pixel in `exture0` at the `v_TexCoord` location, which is the UV coordinate of the fragment currently being processed. In simpler English, `tex2D` returns the color of the pixel of this particular fragment within the texture map. We then take that color and subtract our `ColorMask` from it, so for example if our `ColorMask` is (0,1,1,0), we will remove all the green and blue from the resulting color and set that as our out `Color`. We also set the colors alpha channel based on the `Alpha` value we bound earlier.

There's more...

A top-level view of how rendering occurs might help you understand the shader process. It all starts with the shader program, vertex buffers, texture coordinates, and so on being passed in to the graphics device. Then this information is sent off to a vertex shader, which can then transform that vertex, do lighting calculations and more (we will see this process shortly). The vertex shader is executed once for every vertex and a number of different values can be output from this process (these are the out attributes we saw in the shader earlier). Next the results are transformed, culled, and clipped to the screen, discarding anything that is not visible, then **rasterized**, which is the process of converting from vector graphics to pixel graphics, something that can be drawn to the screen.

The results of this process are fragments, which you can think of as "prospective pixels," and the fragment are passed in to the fragment shader. This is why they are called fragment shaders instead of pixel shaders, although people commonly refer to them using either expression. Once again, the fragment shader is executed once for each fragment. A fragment shader, unlike a vertex shader, can only return a single attribute, which is the RGBA color of the individual pixel. In the end, this is the value that will be displayed on the screen. It sounds like a horribly complex process, but the GPUs have dedicated hardware for performing exactly such operations, millions upon millions of times per second. That description also glossed over about a million tiny details, but that is the gist of how the process occurs.

So if you think back to `Texture.fcg`, think of it as a miniature program that gets run for each potential pixel to be displayed on

your screen. Each fragment (pixel to be) can be passed a great deal of information in the form of uniform values and attributes, but at the end of the day can only make one decision: What color is this pixel going to be?

Note

The PlayStation Mobile SDK makes use of a technology called Cg, which is a specialized programming language created by NVIDIA in collaboration with Microsoft. It has a C like syntax, and applications are run through the Cg compiler, creating a native DirectX or OpenGL shader as a result. In PlayStation Mobile Studio, the compilation process is handled for you during the build process if you set an asset's build action to shader program. This configuration is done automatically when you create a new shader. This process takes fragment shader (`.fcg`) and vertex shader (`.vcg`) files and creates the `.cgx` file you use when creating a `ShaderProgram`.

See also

- This chapter only covers the very basic details of the Cg programming language. More information and additional tools can be found on nVidia's developer site located at http://bit.ly/U9XwxX. Also see the next recipe, *A vertex shader in action*, for more information on shders in PlayStation Mobile.

A vertex shader in action

In this recipe, we create a simple vertex shader that will allow us to move each individual vertex in the cube along its normal.

Getting ready

Load **PlayStation Mobile Studio** and create a new project. This example again builds heavily on the code from the previous tutorials, so the easiest thing would be to just duplicate or edit the prior recipe's project. The complete project is available as `Ch7_Example5`.

How to do it...

1. Edit `AppMain.cs`; I will assume that you are working from the prior recipes source code. For this recipe we are going to also require normal information for our textured cube, so we have to make a small alteration to `MakeCube()`. When we construct our `VertexBuffer`, we need to inform it of the additional normal array; this is done by adding an additional type to the constructor as follows:

```
VertexBuffer vertexBuffer =
new VertexBuffer(24, 36,
VertexFormat.Float3,
VertexFormat.Float2,VertexFormat.Float3);
```

2. Now we create the new array of `Float3` values. The positions array create the following new array:

```
float[] normals = {
     0.0f,  1.0f, 0.0f,
0.0f,  1.0f,  0.0f, 0.0f,
1.0f, 0.0f, 0.0f,  1.0f,
0.0f, //top
     0.0f,  -1.0f, 0.0f,
```

```
0.0f,  -1.0f,  0.0f, 0.0f,
-1.0f, 0.0f, 0.0f,  -1.0f,
0.0f, //top
     0.0f,  0.0f, 1.0f,
0.0f,  0.0f,  1.0f, 0.0f,
0.0f, 1.0f, 0.0f,  0.0f,
1.0f, //front
     0.0f,  0.0f, -1.0f,
0.0f,  0.0f,  -1.0f, 0.0f,
0.0f, -1.0f, 0.0f,  0.0f,
-1.0f, //back
     1.0f,  0.0f, 0.0f,
1.0f,  0.0f,  0.0f, 1.0f,
0.0f, 0.0f, 1.0f,  0.0f,
0.0f, //right
     -1.0f,  0.0f, 0.0f,
-1.0f,  0.0f,  0.0f, -1.0f,
0.0f, 0.0f, -1.0f,  0.0f,
0.0f //left
};
```

3. We now want to make sure our `normals` are added to the `VertexBuffer` with the following code:

```
vertexBuffer.SetVertices(0,
positions);
vertexBuffer.SetVertices(1,
texcoords);
vertexBuffer.SetVertices(2,normals);
vertexBuffer.SetIndices(indices);
```

4. In our `Main`, we now need to bind our additional normal attribute, which is accomplished by doing the following:

```
textureShader.SetAttributeBinding(0,
"a_Position");
textureShader.SetAttributeBinding(1,
"a_TexCoord");
textureShader.SetAttributeBinding(2,
"a_Normal");
float magnitude = 0.0f;
```

5. We also declared the `magnitude` float value, which is going to control how far along its normal a vertex can be moved. Finally, in our event loop, after we get the `camera` matrix, we add the following code:

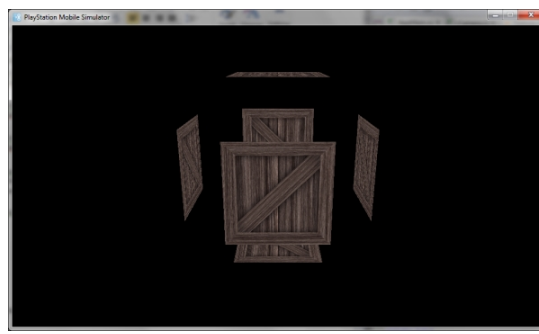
```
if((GamePad.GetData(0).Buttons
& GamePadButtons.Cross) ==
GamePadButtons.Cross)
quit = true;

if((GamePad.GetData(0).Buttons
& GamePadButtons.Square) ==
GamePadButtons.Square){
if(magnitude < 0.5f)
magnitude += 0.01f;
}
if((GamePad.GetData(0).Buttons
& GamePadButtons.Circle) ==
GamePadButtons.Circle){
if(magnitude > 0f) magnitude
-= 0.01f;
}
textureShader.SetUniformValue(textureShac
```

6. Last, we enter the following code for our vertex shader `Texture.vcg`:

```
void main(float4 in
a_Position  : POSITION,
float2 in a_TexCoord  :
TEXCOORD0,
float3 in a_Normal : NORMAL,
float4 out v_Position :
POSITION,
float2 out v_TexCoord :
TEXCOORD0,
uniform float4x4
WorldViewProj,
uniform float Magnitude
   )
```

```
{
v_Position= mul( a_Position,
WorldViewProj );
v_TexCoord  = a_TexCoord;
v_Position +=
mul(float4(a_Normal,1),WorldViewProj)
* Magnitude;
}
```

7. Run the application and you should see output similar to the following screenshot:



Each time you press the Square button (*A* key), the faces of the mesh will explode outward along the faces' normal until they hit the maximum magnitude. The Circle Button (*D* key) will move the faces in the opposite direction.

You may notice that the initial box appears odd. To make backfaces (the generally not visible part of a polygon) visible, we turned face culling off. This odd rendering is the direct result.

How it works...

Our changes start off with the addition of normal information to our vertex buffer. This starts by telling the `VertexBuffer` constructor to expect another array, this one composed of Vector3 objects. We then create our `normals` array by hand. Think of a normal as the direction each vertex is facing. For example, the four vertices that compose the top face of our crate all should face up, which using

our coordinate system is positive Y. If you look at the first four sets of the three floats in our `normals` array, they are all being assigned the value of (0,1,0). This process is then repeated for every other vertex in the buffer. Given that our box is aligned around the origin, the math itself is pretty straightforward; if the cube wasn't aligned so nicely, the vertices composing our faces certainly wouldn't be so consistent! Now that our `vertexBuffer` knows to expect an additional array, and we've constructed our array, we simply assign it using the `SetVertices()` method, this time passing in the index value of `2`.

Our main is almost identical to the previous recipe, although this time we are binding the variable magnitude to the uniform shader value Magnitude. We also needed to set an attribute binding, which we perform with the call `SetAttributeBinding(2, "a_Normal")`, which is simply saying set the attribute `a_Normal` to the buffer at index position 2. The remaining changes are just some input code, so if the user hits the Square button we increment the magnitude by 0.01f, while if the user hits the Circle button, we decrement it by 0.01f. All the remaining logic is performed in the vertex shader.

Looking at our `Texture.vcg` vertex shader, you will notice that there is one `in` attribute for each of the arrays we bound with `SetAttributeBinding()` calls, `a_Position`, `a_TexCoord`, and `a_Normal`. We also have a pair of `out` values, `v_Position` and `v_TexCoord`, which are the ultimate results of running this vertex shader. `v_Position` holds the transformed position of this particular vertex, keeping in mind that this shader is run once for each vertex. `v_TexCoord` holds the 2D texture coordinate that we passed in; we simply pass this value through. We also have a pair of uniform variables, one for the `WorldViewProjection` matrix and the other for the magnitude to move by. In pretty much every vertex shader you will want to apply the `WorldViewProjection` matrix against each vertex, causing each one to be properly transformed. The name `uniform` might seem a bit scary, but it really just means that it is a constant, in that its value will stay uniform.

Additionally, in this particular shader, we want to translate each vertex along its normal by the `Magnitude` amount. This is handled

by transforming the vertex's normal by the `WorldViewProj` matrix and multiplying the result by the magnitude. We then add this calculated value to the `v_Position float4`, which moves the vertex in its normal direction by the amount specified by magnitude. Like `tex2D`, `mul` is a function built in to the Cg programming language.

There's more...

Unlike a fragment shader, which can only output a single value, `COLOR`, you can output multiple values from a vertex shader. You may have noticed that in both vertex and pixel shaders each parameter ended with a value such as `POSITION` or `TEXCOORD0`. These are called semantics and I will let NVIDIA explain them:

Semantics are, in a sense, the glue that binds a Cg program to the rest of the graphics pipelie. The semantics POSITION and COLOR indicate the hardware resource that the respective member feeds when the Cg program returns its output structure.

Essentially, it is through semantics that we tie a Cg script value back to the actual graphics hardware. There are a number of restrictions on what semantics can be used with each part of the process in PlayStation Mobile. These limits are as follows:

- Semantics that can be used as input for vertex shaders:
    - `POSITION`
    - `NORMAL`
    - `TEXCOORD0-7`
    - `COLOR0-1`
    - `DIFFUSE`
    - `SPECULAR`
    - `TANGENT`
    - `BINORMAL`
    - `BLENDWEIGHT`
    - `BLENDINDICES`
    - `FOGCOORD`
    - `POINTSIZE`

- Semantics that can be used as output for vertex shaders:
    - `POSITION` (`HPOS`)
    - `TEXCOORD0-7` (`TEX0-7`)
    - `COLOR0-1` (`COL0-1`)
    - `FOGC` (`FOG`)
    - `PSIZE` (`PSIZ`)
- Semantics that can be used as input for fragment shaders:
    - `TEXCOORD0-7` (`TEX0-7`)
    - `COLOR0-1` (`COL0-1`)
    - `FOGC` (`FOG`)
    - `POINTCOORD`
    - `WPOS` (There is currently a bug with `WPOS` that causes it to render upside down on the PS Vita.)
    - `FACE`
- Semantics that can be usd as output for fragment shaders:
    - `COLOR`

> **Note**
>
> Up until this point, we have combined all of our shaders into a single `.cgx` file by giving them the extensions `.fcg` and `.vcg`. If for some reason, you want your fragment and vertex shaders to be compiled into separate `.cgx` files, instead give them `.fp.cg` and `.vp.cg` extension, respectively.

The Cg programming language and DirectX's HLSL share a common root, so many HLSL shaders and tools will work with Cg, with a small bit of effort.

See also

- In addition to programming Cg shaders by hand, there is an editor available called FxComposer. It is a bit old now, but should still work; just be aware that PlayStation Mobile only supports a subset of Cg, so it isn't guaranteed your shader will work unchanged. You can download FxComposer at http://bit.ly/TgWwbw.

Adding lighting to your scene

If you take a look through the PlayStation Mobile documentation, you may notice there is no functionality in the SDK for lighting. How then do you add lights to your scene? Well, shaders of course! In this recipe we are going to do exactly that.

Getting ready

Load **PlayStation Mobile Studio** and create a new project. Once again, we will use the prior recipe as a base for this recipe. The complete project is available as `Ch7_Example6`.

How to do it...

1. Using `AppMain.cs` from `Example5`, edit `Main` to match the following (the new code is highlighted):

```
Vector3 lightPosition = new
Vector3(0.0f,10.0f,-3.0f);
textureShader.SetUniformValue(textureShad
lightPosition);
Vector4 ambientLightColor =
new
Vector4(0.05f,0.05f,0.05f,1.0f);

bool done = false;
while(!done) {
```

2. Then further down, edit the `event` loop to match the following code snippet:

```
while(!done) {
graphics.Clear ();

var viewWorldProjection =
camera.GetMatrix();
```

```
textureShader.SetUniformValue(0,
ref viewWorldProjection);

if((GamePad.GetData(0).Buttons
& GamePadButtons.Cross) ==
GamePadButtons.Cross)
done = true;

if((GamePad.GetData(0).Buttons
& GamePadButtons.Square) ==
GamePadButtons.Square)
ambientLightColor = new
Vector4(ambientLightColor.R
+0.05f,ambientLightColor.G,ambientLightCo
else
if((GamePad.GetData(0).Buttons
& GamePadButtons.Circle) ==
GamePadButtons.Circle)
ambientLightColor = new
Vector4(ambientLightColor.R-0.05f,ambient
textureShader.SetUniformValue(textureShad
ambientLightColor);

graphics.Enable(EnableMode.Blend);
graphics.SetBlendFunc(new
BlendFunc(BlendFuncMode.Add,BlendFuncFact

graphics.SetShaderProgram(textureShader);
graphics.SetVertexBuffer(0,vertexBuffer);
graphics.SetTexture(0,texture);

graphics.Enable(EnableMode.CullFace,true)
graphics.DrawArrays(DrawMode.Triangles,0,

graphics.SwapBuffers ();
}
```

3. Next, edit `Texture.fcg` to match the following code snippet:

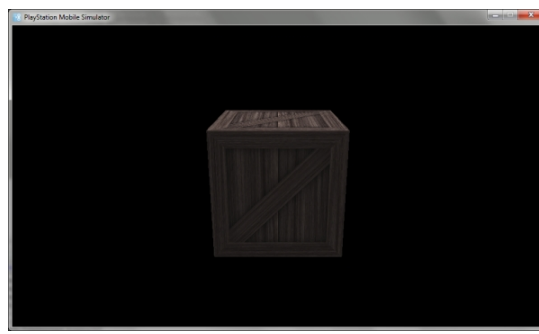
```
void main(
float2 in  v_TexCoord     :
TEXCOORD0,
float3 in v_Light  :
TEXCOORD1,
float3 in v_Normal  :
TEXCOORD2,
float4 out Color  : COLOR,
uniform sampler2D Texture0 :
TEXUNIT0,
uniform float4 ambLightColor
)
{
float4 diffuse = tex2D (
Texture0, v_TexCoord );
Color = ambLightColor +
diffuse *
saturate(dot(v_Light,
v_Normal));
}
```

And `Texture.vcg` as follows:

```
void main(float4 in
a_Position  : POSITION,
float2 in a_TexCoord  :
TEXCOORD0,
float3 in a_Normal  :
NORMAL,
float4 out v_Position :
POSITION,
float2 out v_TexCoord :
TEXCOORD0,
float3 out v_Light :
TEXCOORD1,
float3 out v_Normal :
TEXCOORD2,

uniformfloat3 vecLightPos,
```

```
uniform float4x4
WorldViewProj
   )
{
v_Position= mul( a_Position,
WorldViewProj );
v_TexCoord  = a_TexCoord;
v_Light =
normalize(vecLightPos);
v_Normal = a_Normal;
}
```

4. Now, if you run your application you will see output similar to the following screenshot:



It's our trusty textured crate, this time softly lit by a light source above it. Pressing the Square button (the *A* key) will add some ambient red lighting, while pressing Circle (the *D* key) will reduce it. Once again the *X* button (the *S* Key) will exit.

How it works...

The code within **AppMain.cs** should all be remarkably familiar by now. This time we are setting a `Vector4` color value as a uniform binding that will control the level of ambient light, and starting off with a neutral grey color. We also create and bind a `Vector3` value to hold our light's position in the world. In our event loop, we check for the appropriate button presses and increase or decrease the red component of the light accordingly. The majority of the work is done

in the shaders in this recipe.

First in the vertex shader, we transform the vertex and pass through the texture coordinates as normal. This time we are also passing through the normal information. Finally, we are passing out our light direction in the `v_Light` value.

In our fragment shader we get the starting texel color using the `Tex2D` function and store it as diffuse. We then calculate our final pixel color by adding the diffuse value to the ambient light value and multiplying that value by the saturated value of the dot product of the light vector and the vertex's normal. `Saturate` is another built-in Cg function and it sounds much more complex than it really is. `Saturate` simply makes sure the result is a value between 0 and 1.

There's more...

This example only touches on the basics of the lighting effects that you can create with Cg shaders. Surfaces can be given properties, such as reflectivity; additional light sources can be added, and so on. Check the link in the *See Also* section for some more examples of what you can accomplish creating your lighting using shaders.

See also

The mathematics behind various lighting models is far beyond the scope of this book. There is a very good article on Gamasutra, *Implementing Lighting Models with HLSL*, which can be re-implemented fairly easily in Cg. This article is available at http://bit.ly/XvdeT5.

Using an offscreen frame buffer to take a screenshot

PlayStation Mobile has the ability to render to a user defined offscreen frame buffer. This can be useful for a number of reasons: rendering to a texture, doing post processing before displaying, or, as we will see in this recipe, taking a screenshot.

Getting ready

Load **PlayStation Mobile Studio** and create a new project. This time we are going to be reusing the code we created way back at the start of the chapter during *Creating a Simple 3D scene*. The complete project is available as `Ch7_Example7`.

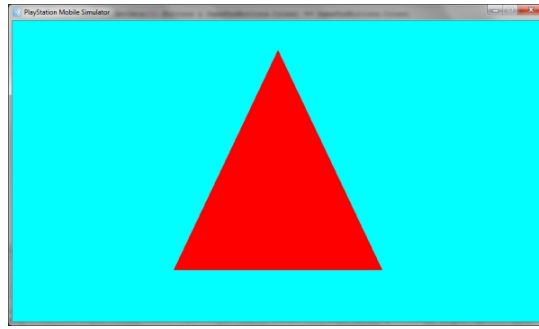
How to do it...

1. Using `AppMain.cs` from the earlier recipe, add the following code to the `main` loop:

```
if((GamePad.GetData(0).Buttons
& GamePadButtons.Start) ==
GamePadButtons.Start){
var frameBuffer = new
FrameBuffer();
var screenshotTexture = new
Texture2D(
graphics.Screen.Width,
graphics.Screen.Height,
false,
PixelFormat.Rgba,
PixelBufferOption.Renderable);
frameBuffer.SetColorTarget(screenshotText
frameBuffer.SetDepthTarget(null);
graphics.SetFrameBuffer(frameBuffer);

var viewProjection =
```

```
projectionMatrix *
viewMatrix;
shaderProgram.SetUniformValue(0,
ref viewProjection);

graphics.SetShaderProgram(shaderProgram);
graphics.SetVertexBuffer(0,vertexBuffer);
graphics.DrawArrays(DrawMode.TriangleStri
graphics.SwapBuffers();

var bytes = new
byte[graphics.Screen.Width *
graphics.Screen.Height * 4];
graphics.ReadPixels(bytes,PixelFormat.Rgb

var image = new
Image(ImageMode.Rgba,new
ImageSize(graphics.Screen.Width,graphics.
image.Export("ScreenShots","demo.jpg");
image.Dispose();

frameBuffer.Dispose();
screenshotTexture.Dispose();
graphics.SetFrameBuffer(null);
}
else {
var viewProjection =
projectionMatrix *
viewMatrix;
shaderProgram.SetUniformValue(0,
ref viewProjection);
graphics.SetShaderProgram(shaderProgram);
graphics.SetVertexBuffer(0,vertexBuffer);
graphics.DrawArrays(DrawMode.TriangleStri

graphics.SwapBuffers ();
}
```

2. Running your application, you will see the output similar to the following screenshot:



It's the red triangle again! This time though, if you press the *Start* button, a screenshot is taken and added to your gallery.

How it works...

In our game loop, we check to see if the user has pressed the Start button. If he or she has not, we render as normal in a rather crude else block. If Start has been pressed, we create a new `FrameBuffer`. Next we create a Texture2D with the same dimensions as the screen. The key thing to notice is the last parameter, `PixelBufferOption.Renderable`, which allows this texture to be written to.

We then set this newly created texture as the rendering target with a call to `SetColorTarget()`. We won't be using a depth target, so we set it to null with a call to `SetDepthTarget`. Next we set our newly created `FrameBuffer` as the active frame buffer by calling `graphics.SetFrameBuffer()`. From this point on, all graphics routines will draw to our offscreen buffer instead of the back buffer for display on screen.

The next few lines of code are exactly the same, setting up the view matrix, setting our shader, and buffer, and rendering the `vertexBuffer`. After we call `SwapBuffers`, our `FrameBuffer` should now contain a capture of the screen. Then it's time to save it as an image.

First we create a byte array large enough to hold the screen, which is 4 bytes per pixel, multiplied by the width and height. We then read

the pixels in to our buffer using `graphics.ReadPixels`. We then create an image of the same dimensions and color depth from this byte buffer. Now that our image is created, we export it, then dispose of it. The `FrameBuffer` class is also an `IDisposable` class, so it needs to be disposed of as well. Finally we set the frame buffer to null in a call to `SetFrameBuffer()`; this sets rendering back to normal.

There's more...

This recipe was mostly for fun and demonstration purposes. The Vita and most Android devices already have built-in screen capture abilities. To take a screenshot using your PlayStation Vita, simply hold the *PS* button and the *Start* button at the same time. Additionally the `ReadPixels` method used in the example is a very expensive operation and should be avoided if possible.

See Also

- See the *Manipulating an image dynamically* recipe in Chapter 1, Getting Started, for more information on working with the `Image` class

Chapter 8. Working with the Model Library

In this chapter we will cover:

- Importing a 3D model for use in PlayStation Mobile
- Loading and displaying a 3D model
- Using BasicProgram to perform texture and shader effects
- Controlling lighting using BasicProgram
- Animating a model
- Handling multiple animations
- Using bones to add a sword to our animated model

Introduction

In the previous chapter we covered working in 3D using PlayStation Mobile. However, we only made use of the simplest primitives, a triangle and a simple textured cube. In a real game you are going to want to make use of more complex shapes, animated and textured models exported from a 3D graphical application such as 3D Studio Max, Maya, or Blender. Fortunately, PlayStation Mobile provides tools for importing such models, as well as a library for loading and displaying them, `Sce.PlayStation.HighLevel.Model`. In addition to handling 3D models, it also provides classes to simplify the use of textures, lights, and shaders.

Importing a 3D model for use in PlayStation Mobile

In this recipe we explore the process of importing a 3D model to the PlayStation Mobile format using the ModelConverter application.

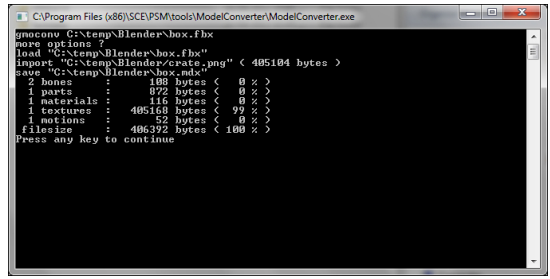
Getting ready

In your favorite 3D modeling application, export your model in one of the formats that PlayStation Mobile supports. Make sure the textures are in the same location and are compatible with the Texture2D supported types. If you have no prior modeling, I have made available a simple textured cube `.Blend` file exported to `.x` format that you can use. It is available as `Ch8_Example1`.

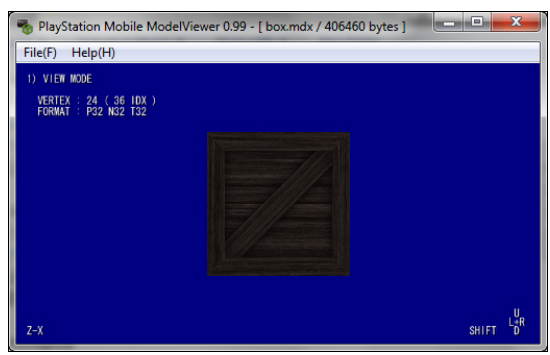
How to do it...

To import a 3D model for use in PlayStation Mobile perform the following step:

1. Open **Windows Explorer** and navigate to the location where `ModelConverter.exe` is located. If you choose the default install and are running a 64-bit copy of Windows, the program will be located at `C:\Program Files (x86)\SCE\PSM\tools\ModelConverter`.
2. Open another Explorer window and navigate to where your 3D model is saved.
3. Now drag the model files you want to convert and drop them on `ModelConverter.exe`. Hold the *Control* key down if you want to specify any command line arguments (see the following screenshot). You can drag and convert multiple files at once. If you held down the *Control* key, you will see something like the following screenshot (otherwise you will see nothing):



4. Your file should now be converted, and there will be a new file of the same name with the `.mdx` extension in the same folder.
5. You can now test how your model will look in PlayStation Mobile by using the utility `ModelViewer.exe`. It is located in the same folder as `ModelConverter.exe`; simply drag the newly generated `.mdx` file and drop it on the **ModelViewer** icon and you should see the following screenshot:



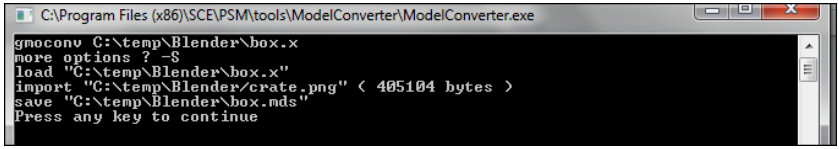
You can rotate the scene using the arrow keys and zoom the camera in and out using the *Z* and *X* keys.

Note that the ModelViewer application requires that the DirectX9 redistributable be installed on your computer.

How it works...

For performance reasons, PlayStation Mobile Studio converts files into its own native format, `.mdx`. Additionally, for debugging reasons you can also export to an ASCII (text) encoded format,

`.mds`. To encode a file as text, hold the *Control* key when dropping the file on `ModelConverter.exe` and when prompted for `more options?` enter `-S`, as shown in the following screenshot:



Of course, if you prefer, ModelConverter can be used completely from the command line. Using ModelConverter from the command line will display additional information about the model conversion process. It supports the following parameters:

- `-o <filename>`: specify output filename
- `-s <scale>`: Scale geometry size
- `-t <scale>`: Scale animation speed
- `-S`: Output in text format
- `-N`: Do not optimize
- `-models`: Merge models into one file
- `-motions`: Merge motions into one file

There's more...

ModelConverter supports the following file formats:

- FBX
- XSI
- COLLADA
- X
- MDX/MDS

If you want to support the XSI format, you need to install Autodesk Crosswalk 2013, available at http://autode.sk/UepCmI.

Note

Model conversion from almost any 3D application is a finicky process. One tip when exporting your model is to immediately re-import it in to your 3D application to see if any problems were

introduced. Some formats certainly work better than others; for example, with Blender, I find that the archaic `.x` format actually produces the best results, but it also is the most restricted. The COLLADA support, however, is awful, although the Blender community is actively working to improve it.

There are a number of different applications you can use to create content and export in the preceding formats. Autodesk has a series of products including 3D Studio Max, Maya, and Softimage, each of which has a several thousand dollar price tag. There are also free options such as Blender or the Wings3D modeler. You can see a fairly comprehensive list of 3D applications at http://bit.ly/SsMD90.

See also

- See **API Overview** | **HighLevel.Model Overview** in the PlayStation Mobile documentation for more details on using ModelConverter as well as limitations of each supported file format

Loading and displaying a 3D model

This recipe demonstrates adding a 3D model to your project as well as the code required to load and display it on screen.

Getting ready

If you haven't already, load **PlayStation Mobile Studio** and create a new project. This example and many that follow are going to make use of a fully textured and animated model that is shipped with the PlayStation Mobile SDK, `walker.mdx`. This is because a great deal of small issues can be introduced when exporting a model from various 3D programs, so it is best to start with a model we know. If you choose the default install, the model file will be available at `C:\Users\Public\Documents\PSM\sample\Model\BasicModelSample\walker.mdx`.

Add the model file to your project, and then be sure to set its **Build Action** to **Content**. You also need to add a reference to the library, `Sce.PlayStation.HighLevel.Model`. The code for this example is available as `Ch8_Example2`.

How to do it...

Open `AppMain.cs` and enter the following code:

```
using System;
using System.Collections.Generic;
using Sce.PlayStation.Core;
using Sce.PlayStation.Core.Graphics;
using Sce.PlayStation.Core.Input;
using
Sce.PlayStation.HighLevel.Model;

namespace Ch8_Example2
```

```
{
  public class AppMain {
    public static void Main (string[]
args) {
      var graphics = new
GraphicsContext ();
      graphics.SetClearColor (255.0f,
255.0f, 255.0f, 0.0f);

      var model = new BasicModel("/
Application/walker.mdx",0);

      Matrix4 view = Matrix4.LookAt(
new Vector3(0.0f,0.0f,25.0f),
                    new
Vector3(0.0f,0.0f,0.0f),
                    Vector3.UnitY);
      Matrix4 proj =
Matrix4.Perspective(FMath.Radians(45),graphics.Sc
      Matrix4 world =
Matrix4.Translation(-
model.BoundingSphere.Xyz);

      model.SetWorldMatrix(ref
world);
      Vector3 ambient = new
Vector3(1.0f,1.0f,1.0f);
      BasicParameters parameters =
new BasicParameters();


parameters.SetProjectionMatrix(ref
proj);
      parameters.SetViewMatrix(ref
view);
      parameters.SetLightAmbient(ref
ambient);

      var program = new
BasicProgram(parameters);
```

```
      bool done = false;
      while(!done) {
        graphics.Clear ();
        var gamepad =
GamePad.GetData(0);
        if((gamepad.Buttons &
GamePadButtons.Cross) ==
GamePadButtons.Cross)
          done = true;
        if((gamepad.Buttons &
GamePaadButtons.Left) ==
GamePadButtons.Left)
          world *=
Matrix4.RotationY(FMath.Radians(1));
        if((gamepad.Buttons &
GamePadButtons.Right) ==
GamePadButtons.Right)
          world *=
Matrix4.RotationY(-FMath.Radians(1));

        model.SetWorldMatrix(ref
world);

        graphics.Enable(
EnableMode.CullFace ) ;
        graphics.SetCullFace(
CullFaceMode.Back,
CullFaceDirection.Ccw ) ;
        graphics.Enable(
EnableMode.DepthTest ) ;
        graphics.SetDepthFunc(
DepthFuncMode.LEqual, true ) ;

        model.Update();
        model.Draw(graphics,program);
        graphics.SwapBuffers ();
      }
      model.Dispose();
      program.Dispose();
```

```
            graphics.Dispose();
        }
    }
}
```

Press *F5* to run your application and you should see the following screenshot:

The model, a man in mid-step, should be drawn on the center of the screen. You can use left and right on the d-pad or the arrow keys to rotate the camera. Press the *X* button (or the *S* key) to exit.

How it works...

The first thing we do is create the required `GraphicsContext` and set the clear color to white. We then load our model file, which is a `BasicModel`. We simply pass the filename in to the constructor. The second parameter represents the index of the model within the `.mdx` file, in case there are multiple models in a single `.mdx` file. Next we set up the `view`, `projection`, and `world` matrices, all of which were described in the previous chapter. We will load the model at the origin and position our camera away 25 units along the z axis. The only change in the process of creating the matrices compared to prior recipes is the following line of code:

```
Matrix4 world = Matrix4.Translation(-
model.BoundingSphere.Xyz);
```

This is because our model is positioned about the origin, in that its feet are located at (0,0,0), meaning we will only see half of the model on screen. Therefore we translate our `world` matrix by the model's `BoundingSphere`, causing it to be centered on the screen. The bounding sphere is the smallest sphere that can contain the entire contents of the model. In addition to using it to determine the model's dimensions, the bounding sphere is also used to perform collision detection.

Next we set the model's location in the 3D world with a call to `SetWorldMatrix`. Then we create a light, a `BasicParameters` object, configure its `view` and `projection` matrices, and pass in our ambient light. Finally, we create a new `BasicProgram` using our newly created `BasicParamters` object. We will cover this process in more detail in the next chapter.

Now we start our game loop. First we clear the back buffer, and then poll the state of the gamepad. We toggle our `done` flag if the user presses the **X** button, and rotate the `world` matrix left or right 1 degree each time the d-pad is pressed in either direction. We then update the model's `world` matrix, in case we rotated the view.

Next we enable culling by calling `Enable( EnableMode.CullFace )`. In the call to `SetCullFace()`, we specify that we want backfaces culled, and that polygons are drawn counterclockwise, which the graphics context uses to determine which direction a polygon is facing. Backface culling basically just removes all polygons that aren't facing the camera, speeding up the rendering process. Next we enable depth testing, again using the `Enable` function. Depth testing will remove all polygons not within the viewing frustum, which we set when we created the `projection` matrix. In this simple example, it doesn't have any real effect on performance, but in a larger scene it certainly will!

Next we call the model's `Update()` method. `Update()` simply applies the model's `world` matrix to all child bones. It is very

important that you call `Update()` for each frame if your model contains any bones, or it will be drawn mangled and not animate properly. Now we tell our model to call `Draw()`, passing in `GraphicsContext` and `BasicProgram` to use them during the rendering process. Like `Update()`, `Draw()` needs to be called once per frame or your model will not be rendered. Finally we present the back buffer to the screen, making our frame visible with a call to `SwapBuffers()`.

After our loop, we clean up our `IDisposable` items. It is important to note that `BasicModel` needs to be disposed off, or it will leak resources.

There's more...

Just like all of the other high-level namespace libraries, the full source code of the `Model` library is available. If you choose the default install, the source code should be located at `C:\Program Files (x86)\SCE\PSM\source\Model` or `C:\Program Files \SCE\PSM\source\Model` if you have a 32-bit version of Windows. If you need to debug into the `Model` source code, you can include this project in your solution instead of adding the library as a reference.

Note

Bones?

Bones are a fundamental part of 3D animation. An animated model is generally composed of a mesh (the polygon data) and a series of bones that manipulate the mesh. For example, a leg mesh might contain three bones, one for the upper leg, one for below the knee, and one for the foot. If you rotate the foot bone, it will then rotate all of the vertices in the mesh that are assigned to that bone, therefore rotating the foot. When you call `Update()` on `BasicModel`, it updates the position of every bone it contains in world space. The coordinates of each bone are relative to the model itself. It is through movement of these bones that animation can occur.

See also

- Refer to the *Creating a simple 3D scene* recipe in Chapter 7, *Into the Third Dimension*, for more details on working in 3D

Using BasicProgram to perform texture and shader effects

In Chapter 7, *Into the Third Dimension*, we applied a number of visual effects on a cube using vertex and fragment shaders. The `Model` library provides the `BasicProgram` class to simplify this process. In this example, we are going to load a 3D crate model and show how to alter its transparency, change its color, and even swap textures completely using `BasicProgram`.

Getting ready

Load **PlayStation Mobile Studio**, create a new project, and add a reference to `Sce.PlayStation.HighLevel.Model`. Instead of building the cube from scratch like we did in Chapter 7, *Into the Third Dimension*, we are going to load it as a model. You can either create your own textured box, or download a working example from http://bit.ly/PtMj9w. The ZIP includes a `.mdx` file of a textured cube, as well as a second texture for the cube. Download and extract the archive, and add `box.mdx` and `crateNeon.png` to your project, setting **Build Action** to **Content** for both of them. You can get the files and complete source code as `Ch8_Example3`.

How to do it...

Open `AppMain.cs` and replace `Main` with the following function:

```
public static void Main (string[]
args) {
  var graphics = new GraphicsContext
();
  graphics.SetClearColor (255.0f,
255.0f, 255.0f, 0.0f);

  var model = new BasicModel("/
Application/box.mdx",0);
```

```
  Matrix4 view = Matrix4.LookAt( new
Vector3(0.0f,0.0f,25.0f),
                 new
Vector3(0.0f,0.0f,0.0f),
                 Vector3.UnitY);
  Matrix4 proj =
Matrix4.Perspective(FMath.Radians(45),graphics.Sc
  Matrix4 world =
Matrix4.Translation(-
model.BoundingSphere.Xyz) *
Matrix4.Scale (3f,3f,3f);

  model.SetWorldMatrix(ref world);

  Vector3 ambient = new
Vector3(1.0f,1.0f,1.0f);
  BasicParameters parameters = new
BasicParameters();

  parameters.SetProjectionMatrix(ref
proj);
  parameters.SetViewMatrix(ref view);
  parameters.SetLightAmbient(ref
ambient);

  var program = new
BasicProgram(parameters);

  BasicTexture texture1 =
model.Textures[0];
  BasicTexture texture2 = new
BasicTexture();
  texture2.FileName = "/Application/
crateNeon.png";
  texture2.Texture = new Texture2D("/
Application/crateNeon.png",false);


  bool done = false;
```

```
  float opacity = 1f;
  Vector3 ambientColor = new
Vector3(1,1,1);

  while(!done) {
    graphics.Clear ();
    var gamepad = GamePad.GetData(0);
    if((gamepad.Buttons &
GamePadButtons.Cross) ==
GamePadButtons.Cross)
      done = true;
    if((gamepad.Buttons &
GamePadButtons.Left) ==
GamePadButtons.Left)
      world *=
Matrix4.RotationY(FMath.Radians(1));
    if((gamepad.Buttons &
GamePadButtons.Right) ==
GamePadButtons.Right)
      world *= Matrix4.RotationY(-
FMath.Radians(1));

    if((gamepad.ButtonsUp &
GamePadButtons.Circle) ==
GamePadButtons.Circle) {
      if(model.Textures[0].FileName
== "/Application/crateNeon.png")
        model.Textures[0] = texture1;
      else
        model.Textures[0] = texture2;
    }
    if((gamepad.Buttons &
GamePadButtons.Up) ==
GamePadButtons.Up) {
      if(opacity < 1f) opacity+=
0.01f;
    }
    if((gamepad.Buttons &
GamePadButtons.Down) ==
GamePadButtons.Down) {
```

```
      if(opacity > 0f) opacity-=
0.01f;
    }
    if((gamepad.ButtonsUp &
GamePadButtons.Triangle) ==
GamePadButtons.Triangle) {
      Random rand = new Random();

      ambientColor = new
Vector3(rand.Next(0,255)/255f,rand.Next(0,255)/25

    }
    model.SetWorldMatrix(ref world);
    model.Materials[0].Ambient =
ambientColor;
    model.Materials[0].Opacity =
opacity;


graphics.Enable(EnableMode.Blend);

graphics.SetBlendFunc(BlendFuncMode.Add,
BlendFuncFactor.SrcAlpha,
BlendFuncFactor.OneMinusSrcAlpha);

graphics.Enable(EnableMode.CullFace);

graphics.SetCullFace(CullFaceMode.Back,
CullFaceDirection.Ccw);

graphics.Enable(EnableMode.DepthTest);

graphics.SetDepthFunc(DepthFuncMode.LEqual,
true);

    model.Update();
    model.Draw(graphics,program);

    graphics.SwapBuffers ();
  }
```

```
            model.Dispose();
            program.Dispose();
            graphics.Dispose();
        }
```

Hit *F5* to run the application and you will see the following screenshot:

A cube will be centered on the screen. Press left or right on the d-pad to rotate the cube and press the **X** button to exit. Press the triangle button to randomize the cube's color, press up or down on the d-pad to increase or decrease the opacity, or press the circle button to toggle between the normal and neon textures.

How it works...

This recipe basically starts off with initialization code identical to the previous recipe, except the clear color is now white and we are loading a different `.mdx` file into our `BasicModel`. After creating our various `view` matrices, we create a `BasicParamaters` object and use it to create a `BasicProgram`. A `BasicProgram` is a preconfigured and substantially easier to use `ShaderProgram`, coupled with a prewritten vertex and fragment shader. After creating our program, we take a reference to our model's existing texture

called `texture1`, as well as declaring a new `BasicTexture`, `texture2`. We then assign `texture2` a filename and load that file into the `Texture` member of `texture2`. We then declare our exit flag `done`, as well as declare a variable to hold the texture's opacity and a `Vector` to hold the texture's color.

In our game loop, we clear the screen then check the gamepad for input. If the user presses the **X** button, we set the `done` flag and cause the application to exit. Once again, pressing left and right on the d-pad causes the scene to rotate. We then check to see if the circle button has been pressed and released, and if it has, we swap the active texture on the model to either `texture1` or `texture2`. Next we check to see if the user pressed up or down on the d-pad, and if so we increase or decrease the opacity value. Lastly we check to see if the user pressed the triangle button, and if he/she has we assign `ambientColor` a random red/green/blue value. This color value is a floating point value between 0 and 1, while the `Random` object returns an `int` value, so we simply request a random value between 0 and 255 and divide that value by 255, giving us a value in the proper range of 0 to 1.

Now we update the model's `world` matrix, and set the texture's `Ambient` and `Opacity` values. All of the remaining code is identical to the prior recipe except for the fact that we add an `Enable()` call to enable the `Blend` mode. If you have transparent or translucent objects in your scene, you need to enable blending. See the link in the *See also* section of this recipe for more details on how blending works in OpenGL.

There's more...

If you have more than one `BasicProgram`, or simply do not want to pass it in to the model's `Draw()` method, the `model` class has a method named `BindPrograms()` that you can pass `BasicProgramContainer` in to. `BasicProgramContainer` is a simple data type that holds a collection of key/value pairs of `BasicPrograms`. When creating `BasicProgramContainer`, you can pass in a `BasicParameters` value to the constructor and those parameters will be shared between all contained `BasicPrograms`. If you do not want to share a `BasicParamters`

value, simply pass in null. The following is an example:

```
var param = new BasicParameters();
var container = new
BasicProgramContainer(param);
var progam = new BasicProgram();
container.Add("program1",prog);
model.BindPrograms(container);
```

As mentioned earlier, `BasicProgram` is a wrapper around the vertex and fragment shader functionality. Behind the scenes `BasicProgram` is populating a vertex and fragment shader for you. If you want to extend the functionality of these shaders or `BasicProgram`, you will need to build the `Model` library yourself. Assuming a default install, the shaders used by `Model` are located at `C:\Program Files (x86)\SCE\PSM\source\Model\shaders`, in a pair of files named `Basic.fcg` and `Basic.vcg`

See also

- See a fragment (pixel) shader in action and a vertex shader in action for an example of performing the same process without the help of the `Model` library. Additionally, see http://bit.ly/X9hni7 (OpenGL site: Transparency, Translucency, and Blending) for more details on GL blending and blending functions.

Controlling lighting using BasicProgram

In this recipe we will look at how the `Model` library simplifies the process of adding lighting to our scene.

Getting ready

In Studio create a new project and add a reference to the `Sce.PlayStation.HighLevel.Model` library. Also add `walker.mdx` and set its **Build Action** to **Content**. The project can be downloaded as `Ch8_Example4`.

How to do it...

Open `AppMain.cs` and enter the following in place of the `Main` function:

```
public static void Main (string[]
args) {
  var graphics = new GraphicsContext
();
  graphics.SetClearColor (0.0f, 0.0f,
0.0f, 0.0f);

  var model = new BasicModel("/
Application/walker.mdx",0);

  Matrix4 view = Matrix4.LookAt( new
Vector3(0.0f,0.0f,25.0f),
                new
Vector3(0.0f,0.0f,0.0f),
                Vector3.UnitY);
  Matrix4 proj =
Matrix4.Perspective(FMath.Radians(45),graphics.Sc
  Matrix4 world =
```

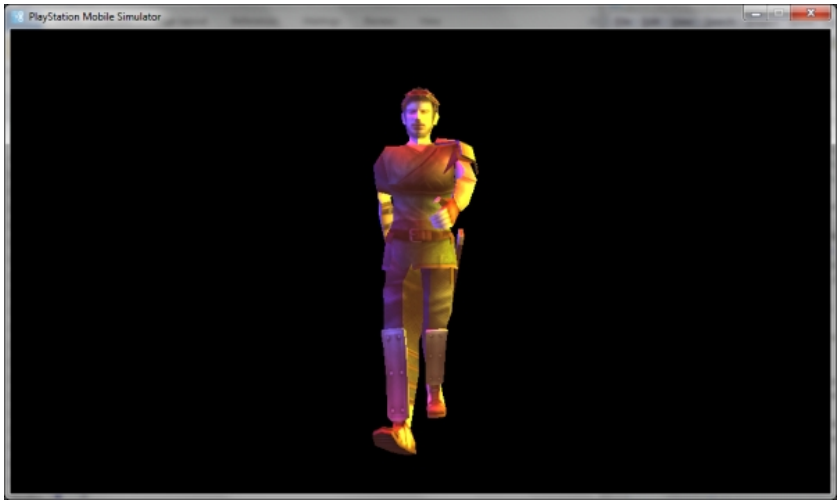
```
Matrix4.Translation(-
model.BoundingSphere.Xyz);
  model.SetWorldMatrix(ref world);
  var program = new BasicProgram();


  bool done = false;
  bool light1On, light2On, light3On,
fogOn, ambientOn;
  light1On = light2On = light3On =
fogOn = ambientOn = false;
  Vector3 light1Diffuse,
light2Diffuse, light3Diffuse,
fogColor, ambient;
  Vector3 light1Direction,
light2Direction, light3Direction;

  while(!done) {
    BasicParameters parameters =
program.Parameters;

parameters.Enable(BasicEnableMode.Fog,true);

parameters.Enable(BasicEnableMode.Lighting,true);
    graphics.Clear();

    var gamepad = GamePad.GetData(0);
    if((gamepad.Buttons &
GamePadButtons.Cross) ==
GamePadButtons.Cross)
      done = true;
    if((gamepad.Buttons &
GamePadButtons.Left) ==
GamePadButtons.Left)
      world *=
Matrix4.RotationY(FMath.Radians(1));
    if((gamepad.Buttons &
GamePadButtons.Right) ==
GamePadButtons.Right)
      world *= Matrix4.RotationY(-
```

```
FMath.Radians(1));

    if((gamepad.ButtonsUp &
GamePadButtons.Triangle) ==
GamePadButtons.Triangle)
      light1On = !light1On;
    if((gamepad.ButtonsUp &
GamePadButtons.Circle) ==
GamePadButtons.Circle)
      light2On = !light2On;
    if((gamepad.ButtonsUp &
GamePadButtons.Square) ==
GamePadButtons.Square)
      light3On = !light3On;
    if((gamepad.ButtonsUp &
GamePadButtons.Start) ==
GamePadButtons.Start)
      fogOn = !fogOn;
    if((gamepad.ButtonsUp &
GamePadButtons.Select) ==
GamePadButtons.Select)
      ambientOn = !ambientOn;
    light1Direction = new
Vector3(0f,-3f,0f);
    if(light1On)
      light1Diffuse = new
Vector3(1f,0f,0f);
    else
      light1Diffuse = new
Vector3(0f,0f,0f);

    light2Direction = new
Vector3(-3f,0f,0f);
    if(light2On)
      light2Diffuse = new
Vector3(1f,1f,0f);
    else
      light2Diffuse = new
Vector3(0f,0f,0f);
```

```
    light3Direction = new
Vector3(3f,0f,0f);
    if(light3On)
      light3Diffuse = new
Vector3(0f,0f,1f);
    else
      light3Diffuse = new
Vector3(0f,0f,0f);
    if(fogOn)
      fogColor = new
Vector3(1.0f,0.4f,0.3f);
    else
      fogColor = new
Vector3(0f,0f,0f);

    if(ambientOn)
      ambient = new
Vector3(1f,1f,1f);
    else
      ambient = new
Vector3(0f,0f,0f);

    model.SetWorldMatrix(ref world);


parameters.SetProjectionMatrix(ref
proj);
    parameters.SetViewMatrix(ref
view);
    parameters.SetLightAmbient(ref
ambient);
    parameters.SetLightCount(3);


parameters.SetLightDirection(0,ref
light1Direction);
    parameters.SetLightDiffuse(0,ref
light1Diffuse);

parameters.SetLightDirection(1,ref
```

```
light2Direction);
    parameters.SetLightDiffuse(1,ref
light2Diffuse);

parameters.SetLightDirection(2,ref
light3Direction);
    parameters.SetLightDiffuse(2,ref
light3Diffuse);
    parameters.SetFogColor(ref
fogColor);
    parameters.SetFogRange(0f,100f);

graphics.Enable(EnableMode.Blend);

graphics.SetBlendFunc(BlendFuncMode.Add,
BlendFuncFactor.SrcAlpha,
BlendFuncFactor.OneMinusSrcAlpha);

graphics.Enable(EnableMode.CullFace);

graphics.SetCullFace(CullFaceMode.Back,
CullFaceDirection.Ccw);

graphics.Enable(EnableMode.DepthTest);

graphics.SetDepthFunc(DepthFuncMode.LEqual,
true);

    model.Update();
    model.Draw(graphics,program);

    graphics.SwapBuffers ();
  }
  model.Dispose();
  program.Dispose();
  graphics.Dispose();
}
```

Run the application and you should see the following screenshot:



Initially the screen will be completely dark, until you turn on some lights. Once again, press left and right on the d-pad to rotate the scene and click on **X** to exit. Hitting the triangle button turns on a directional light from above the character, hitting the square button turns on a light on the left-hand side, while hitting circle turns on a light on the right-hand side. Hitting the **Select** button toggles the ambient lighting, while the **Start** button turns fog on and off.

How it works...

This recipe starts off virtually identically to prior recipes in this chapter. We create the graphics context, set our clear color to black, load the model, and then create and configure our three matrices. This time we create `BasicProgram` without specifying parameters, as those are going to change each pass through the loop. We then create a `bool` to control when our application is done and one `bool` for each configurable item in the scene. We then create a `Vector3` to hold the color of each of our three diffuse lights, the fog, and the ambient lighting. Then we create `Vector3` objects to hold the direction of each of the diffuse lights.

In our game loop, we take a reference to the program's `Parameters` value; then enable fog and lighting. We then check the state of the gamepad and toggle features on or off depending on which buttons are pressed. Then for each light, we set the direction

(`light1` is aiming down the y axis, `light2` is aiming left along the x axis, and `light3` is aiming right along the x axis), then depending on whether the light is on or off, we set its color. If a light is off, we set the color to (0,0,0). If the light is on, we set it to a particular color (`light1` is red, `light2` is green, and `light3` is blue). We then perform a similar process for the ambient light and fog, minus the positioning.

We need to tell the program how many lights are in the scene, which we accomplish with the `SetLightCount` method. Then for each diffuse light, we call `SetLightDirection()` and `SetLightDiffuse()`. The first parameter is the index of the light, while the second parameter is the direction and color, respectively. Next we pass our fog color in by calling `SetFogColor`, and set the near and far depth ranges of the fog with a call to `SetFogRange()`. All of the remaining code is identical to the prior recipe.

There's more...

When discussing lighting, we've seen a few words over and over such as ambient, specular, and diffuse. An ambient light is a directionless light; it is the general pervasive light in a scene. Diffuse and specular lights, however, both have a direction—think of a spot light or laser beam. The difference between diffuse and specular lights is how the light is reflected from a surface. Diffuse light scatters equally in all directions after hitting a surface, while a specular light tends to reflect in a specific direction. For a general directional light, such as a lamp or spot light, it is generally a diffuse light you want. You can also make a model emit its own light by setting the `Emission` property. `BasicProgram` supports up to three lights (via `SetLightCount`), in addition to a single ambient light. If you want to use more lights, you will need to extend the shader yourself.

There is one major limitation to the current lighting solution provided by `BasicProgram`. You currently cannot give a light a position, only a direction. By increasing the magnitude of the light's direction vector you can increase the intensity of the light—for example a light with the direction (0,-5,0) will cast a light down the y axis that is

much more intense than a light with a direction of (0,-1,0)—but you cannot set the light's source position, making localized direct lights such as a lamp impossible to implement. Adding support in a Cg shader for additional lights or a spotlight wouldn't be too difficult; see the link in the *See also* section of this recipe for an example of creating a spotlight using Cg.

See also

- See the *Adding lighting to your scene* recipe in Chapter 7, *Into the Third Dimension*, for an example of using vertex and fragment shaders to perform lighting. Also see http://bit.ly/RAUb7t (NVIDIA Cg Tutorial: Lighting) for more details on lighting in Cg shaders, as well as an example shader for implementing a spot light.

Animating a model

This recipe will show how easy it is to animate a model using the `HighLevel.Model` library.

Getting ready

Create a new project, add `Walker.mdx`, and set the **Build Action** to **Content**. The code and all materials for this recipe are available as `Ch8_Example5`.

How to do it...

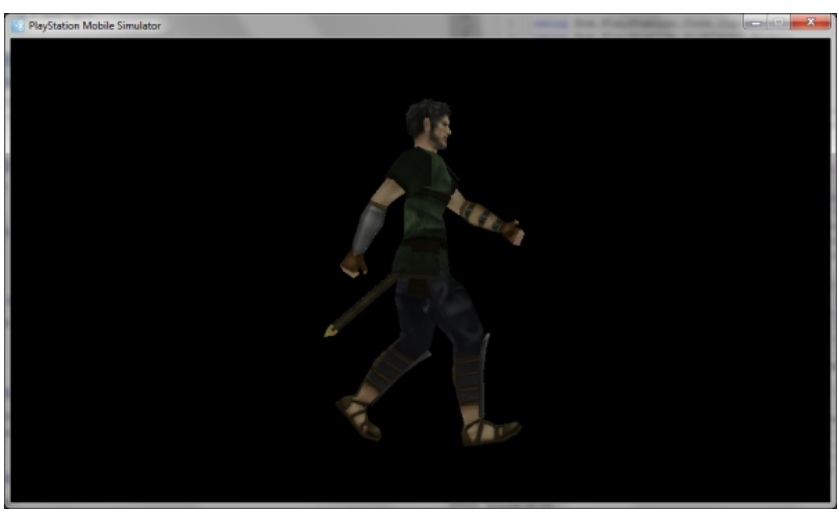
Edit `AppMain.cs` and edit `Main` to match the following (changes highlighted):

```
public static void Main (string[]
args) {
  var graphics = new
GraphicsContext();
  graphics.SetClearColor (0.0f, 0.0f,
0.0f, 0.0f);

  var model = new BasicModel("/
Application/walker.mdx",0);

  Matrix4 view = Matrix4.LookAt( new
Vector3(0.0f,0.0f,25.0f),
                new
Vector3(0.0f,0.0f,0.0f),
                Vector3.UnitY);
  Matrix4 proj =
Matrix4.Perspective(FMath.Radians(45),graphics.Sc
  Matrix4 world =
Matrix4.Translation(-
model.BoundingSphere.Xyz) *
Matrix4.RotationY(FMath.Radians(90f));
```

```
  model.SetWorldMatrix(ref world);

  BasicParameters parameters = new
BasicParameters();
  parameters.SetProjectionMatrix(ref
proj);
  parameters.SetViewMatrix(ref view);
  var ambient = new
Vector3(1f,1f,1f);
  parameters.SetLightAmbient(ref
ambient);

  var program = new
BasicProgram(parameters);
  bool done = false;

  System.Diagnostics.Stopwatch
stopWatch = new
System.Diagnostics.Stopwatch();
  stopWatch.Start();

  long previousTicks = 0;
  while(!done) {
    float delta =
(stopWatch.ElapsedTicks -
previousTicks) /
(float)System.Diagnostics.Stopwatch.Frequency;
    previousTicks =
stopWatch.ElapsedTicks;
    graphics.Clear();

    var gamepad = GamePad.GetData(0);
    if((gamepad.Buttons &
GamePadButtons.Cross) ==
GamePadButtons.Cross)
      done = true;

    model.SetWorldMatrix(ref world);
    graphics.Enable(
```

```
EnableMode.CullFace );
    graphics.SetCullFace(
CullFaceMode.Back,
CullFaceDirection.Ccw );
    graphics.Enable(
EnableMode.DepthTest );
    graphics.SetDepthFunc(
DepthFuncMode.LEqual, true );

    model.Update();
    model.Animate(delta);
    model.Draw(graphics,program);

    graphics.SwapBuffers ();
  }
  model.Dispose();
  program.Dispose();
  graphics.Dispose();
}
```

Hit *F5* to run the application and you should see the following screenshot:



A side view of the walker mesh will be drawn centered on the screen; this time it will live up to its name and walk! You can still

rotate the camera using the d-pad and exit the application by pressing the **X** button.

How it works...

Since this recipe is so similar to the prior recipe, I will only address the highlighted changes. The first difference is that we apply an additional 90-degree rotation to our `world` matrix; this is simply for us to see a slightly more pleasing side view of our walker by default. Then right before our event loop we create a `StopWatch` object and start it running. We then declare a `long` object that will hold the tick count of the previous pass through the game loop.

In the game loop we calculate how much time has occurred since the last pass through the loop. We obtain this value by subtracting the current `ElapsedTicks` from the elapsed ticks from the previous pass through the loop, then divide this value by the `StopWatch Frequency`. The frequency is the resolution that the timer is running at, and is a value representing the number of "ticks" in a second. At this point, our delta value holds the number of seconds that occurred since the last loop occurred.

The only remaining difference is that after we call `Update()` and before we call `Draw()` (this order is very important!), we call `Animate()`, passing it the amount of time in seconds to advance the animation by. This will cause the animation to advance consistently regardless of the speed of the underlying hardware.

There's more...

Passing a negative value to `Animate()` will play the animation in reverse. As you will see shortly, you can have multiple animations in the same model.

See also

- See the *Handling multiple animations* recipe for details on how you deal with more than one animation in a single `.mdx` file

Handling multiple animations

This recipe illustrates how to handle multiple animations in a single model.

Getting ready

This recipe requires an MDX file with multiple animations, which unfortunately `walker.mdx` does not have and none of the other examples provided with the SDK will work, for reasons explained next. Therefore I acquired a fully rigged model from the Internet and created my own simple animations using Blender. The model itself is from OpenGameArt and can be downloaded from http://bit.ly/QeoPGP. The MDX file I generated is included along with all of the source code as `Ch8_Example6`.

Create a new project, add the preceding `.mdx` file to the project, and set its **Build Action** to **Content**. You also need to add a reference to `Sce.PlayStation.HighLevel.Model`.

How to do it...

Open `AppMain.cs` and replace the `Main` function with the following (once again, changes are bolded):

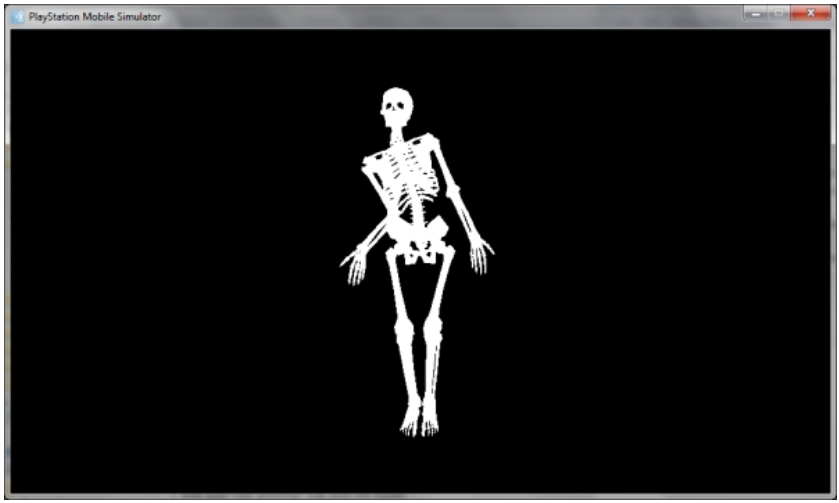
```
public static void Main (string[]
args) {
  var graphics = new GraphicsContext
();
  graphics.SetClearColor (0.0f, 0.0f,
0.0f, 0.0f);
  var model = new BasicModel("/
Application/fgc_skeleton.mdx",0);
  Matrix4 view = Matrix4.LookAt( new
Vector3(0.0f,0.0f,60.0f),
                new
Vector3(0.0f,0.0f,0.0f),
```

```
                    Vector3.UnitY);
  Matrix4 proj =
Matrix4.Perspective(FMath.Radians(45),graphics.Sc
  Matrix4 world =
Matrix4.Translation(0,-20f,0) *
Matrix4.RotationY(FMath.Radians(45f));

  model.SetWorldMatrix(ref world);

  Vector3 ambient = new
Vector3(0.3f,0.3f,0.3f);
  BasicParameters parameters = new
BasicParameters();

  parameters.SetProjectionMatrix(ref
proj);
  parameters.SetViewMatrix(ref view);
  parameters.SetLightAmbient(ref
ambient);

  var program = new
BasicProgram(parameters);

  model.SetCurrentMotion(0);

  var frameRate = 30f;
  bool done = false;
  Vector3 ambientColor = new
Vector3(1,1,1);
  while(!done) {
    graphics.Clear ();
    var gamepad = GamePad.GetData(0);
    if((gamepad.Buttons &
GamePadButtons.Cross) ==
GamePadButtons.Cross)
      done = true;
    if((gamepad.Buttons &
GamePadButtons.Left) ==
GamePadButtons.Left)
      world *=
```

```
Matrix4.RotationY(FMath.Radians(1));

    if((gamepad.Buttons &
GamePadButtons.Right) ==
GamePadButtons.Right)
      world *= Matrix4.RotationY(-
FMath.Radians(1));
    if((gamepad.Buttons &
GamePadButtons.R) ==
GamePadButtons.R) {
      frameRate += 1f;

model.Motions[model.CurrentMotion].FrameRate
= frameRate;
    }

    if((gamepad.Buttons &
GamePadButtons.L) ==
GamePadButtons.L) {
      frameRate -= 1f;

model.Motions[model.CurrentMotion].FrameRate
= frameRate;
    }

    if((gamepad.ButtonsUp &
GamePadButtons.Up) ==
GamePadButtons.Up) {
      model.SetCurrentMotion(0,1f);
    }
    if((gamepad.ButtonsUp &
GamePadButtons.Down) ==
GamePadButtons.Down) {
      model.SetCurrentMotion(1,1f);
    }

    model.SetWorldMatrix(ref world);


graphics.Enable(EnableMode.Blend);
```

```
graphics.SetBlendFunc(BlendFuncMode.Add,
BlendFuncFactor.SrcAlpha,
BlendFuncFactor.OneMinusSrcAlpha);

graphics.Enable(EnableMode.CullFace);

graphics.SetCullFace(CullFaceMode.Back,
CullFaceDirection.Ccw);

graphics.Enable(EnableMode.DepthTest);

graphics.SetDepthFunc(DepthFuncMode.LEqual,
true);

    model.Update();
    model.Animate(0.01f);
    model.Draw(graphics,program);

    graphics.SwapBuffers ();
  }
  model.Dispose();
  program.Dispose();
  graphics.Dispose();
}
```

Run the application and hopefully you will see the following:



A shrugging skeleton will be drawn on the center of the screen. Pressing up and down on the d-pad will toggle between the two available animations, shrug and lean. Pressing the right bumper (*E* key) will increase the animation speed, while pressing the left bumper (*Q* key) will decrease it. Once again, rotate the scene to the left and right and press the **X** button to exit.

How it works...

Once again, this recipe shares a great deal of common code with recipes we covered previously, so we will only cover the changes. The first obvious change is that we load our model from the `fgc_skeleton.mdx` file. We make a few changes to the `view` and `world` matrices to better accommodate the dimensions of our skeleton. We set the model playing its first animation by calling `SetCurrentMotion()`, passing the index of the animation to play. We then declare a variable to hold our `frameRate`, which is the rate we want the animation to play back at.

In our game loop, we check to see if the right or left bumper is pressed, and if so we increase or decrease `frameRate` by 1, depending on which was selected. We then update the `FrameRate` property of the current animation to our newly updated `frameRate` value. As you can see, the model contains an array named `Motions` containing all of its animations, as well as a

`CurrentMotion` property holding the index of the currently playing motion. You can completely stop an animation by setting `FrameRate` to 0. Each `Motion` contains more information, including the starting frame, ending frame, and current frame, as well as the `FrameRepeat` property, which will determine if an animation will loop when completed.

If the user presses up on the d-pad, we set Motion 0 as the active motion, while if the user presses down, we set `Motion 1` as the active motion. This is accomplished by calling `SetCurrentMotion()`. The first parameter is the index of `Motion` set to active, while the second value is the number of seconds to delay before transitioning. It will default to 0 if you do not provide a value. Once again we update our animation by calling `Animate()`. This time, to simplify the recipe, I pass in a value of a tenth a second. This is not ideal, as the animation will advance at different rates on different hardware.

There's more...

Note

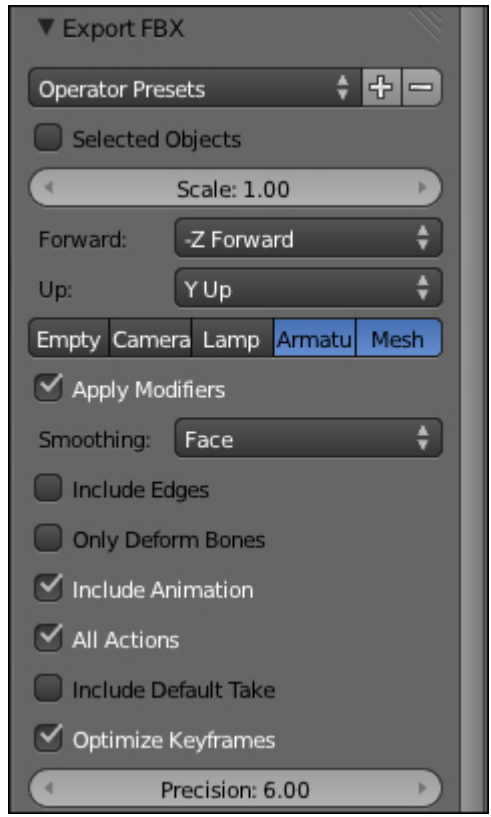
When setting out to write this chapter I had a bit of a challenge. The `walker.mdx` file only has a single animation. Therefore I considered co-opting one of the MDX files from the various demos included with the SDK, such as the hero model from the RPG demo. That is when I discovered that these MDX files are *not* compatible with the MDX files required by the `Model` library. In fact, only a single sample and none of the demos make use of the `Model` library.

Getting multiple animations out of Blender proved to be somewhat of a trick. I managed to create an MDX file with multiple animations using **Action Editor** in the **DopeSheet** menu to create each animation, but once imported they never worked correctly and only a single animation played using the ModelViewer. I did, however, manage to get multiple animations to work using the following process:

1. In Blender, model each animation in a separate `.Blend`

file.

2. Export each file as an Autodesk FBX. When exporting, select only the **Armature** and **Mesh** tabs; make sure that **Include Animation** and **Optimize Keyframes** are checked, and **Include Default Take** is not. Your export setting should look like the following screenshot:

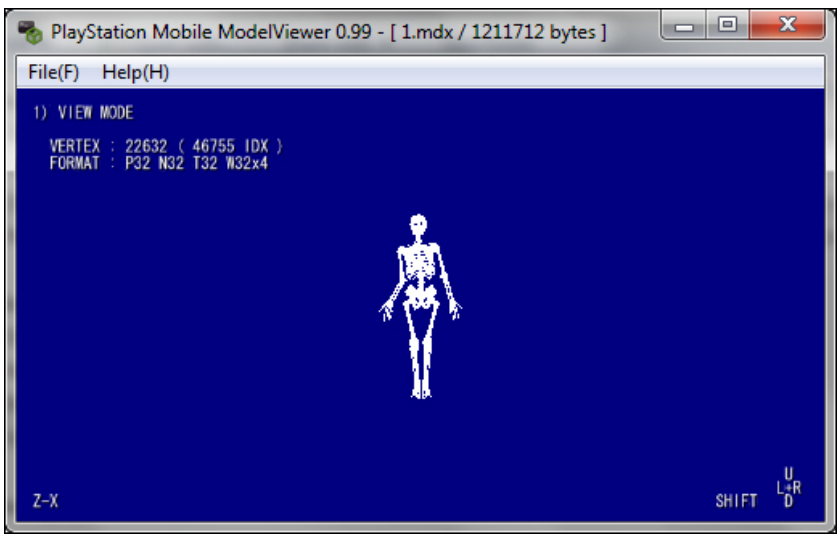


3. Perform the preceding process for each of your animations. In **Windows Explorer**, while keeping the *Ctrl* key pressed, select all of the FBX files and drag them onto `ModelConverter.exe`.
4. When prompted for params, enter `-motions`.

This is the process used to create the animations in this recipe and will result in a single MDX file with multiple animations that will work with PlayStation Mobile.

When working on this chapter, I had a bit of a battle getting my animations work properly. While debugging I discovered that the ModelViewer should be your new best friend. Also, when importing an external file into Studio over and over, you will save a great deal of time adding it as a link instead of adding the file directly to the project.

The following screenshot shows ModelViewer:



ModelViewer is a simple utility that is located in the same directory as ModelConverter. You can use it to preview how your MDX file will look once imported into your application. Simply drag-and-drop the MDX file on `ModelViewer.exe` to load your model. The *Z* and *X* keys zoom your camera in and out, while the arrow keys rotate the project. Pressing a number key from *1* through *5* will toggle the application between different modes, as follows:

- **View Mode**
- **Animation**
- **State Switches** (up and down arrows to navigate, left and right to toggle)
- **Information**
- **Help**

ModelViewer is invaluable for troubleshooting model importing and

rendering problems.

See also

- See the *Animating a model* recipe for details on how to properly advance the time step passed to the `Animate` function

Using bones to add a sword to our animated model

One very common action to perform in 3D games is to dynamically add weapons and details to a mesh. This recipe shows the process of loading a sword and binding it to the walker model's wrist.

Getting ready

Create a new project, add the `walker.mdx` model, and set its **Build Action** to **Content**. Be sure to add a reference to the `Sce.PlayStation.HighLevel.Model` library. I modeled an extremely simple sword mesh in Blender and exported it to MDX as `sword.mdx`. Add it to your project and again set its **Build Action** to **Content**. It is available with the source code as `Ch8_Example7`.

How to do it...

Once again this code shares a great deal with earlier recipes. Open `AppMain.cs` and change `Main` to match the following code (bolded area represents changes):

```
public static void Main (string[]
args) {
  var graphics = new GraphicsContext
();
  graphics.SetClearColor (0.0f, 0.0f,
0.0f, 0.0f);

  var model = new BasicModel("/
Application/walker.mdx",0);
  var sword = new BasicModel("/
Application/sword.mdx",0);

  Matrix4 view = Matrix4.LookAt( new
Vector3(0.0f,0.0f,25.0f),
```

```
                new
Vector3(0.0f,0.0f,0.0f),
                Vector3.UnitY);
  Matrix4 proj =
Matrix4.Perspective(FMath.Radians(45),graphics.Sc
  Matrix4 world =
Matrix4.Translation(-
model.BoundingSphere.Xyz);

  model.SetWorldMatrix(ref world);

  Vector3 ambient = new
Vector3(1.0f,1.0f,1.0f);
  BasicParameters parameters = new
BasicParameters();

  parameters.SetProjectionMatrix(ref
proj);
  parameters.SetViewMatrix(ref view);
  parameters.SetLightAmbient(ref
ambient);

  var program = new
BasicProgram(parameters);

  bool done = false;
  Vector3 ambientColor = new
Vector3(1,1,1);
  while(!done) {
    graphics.Clear ();
    var gamepad = GamePad.GetData(0);
    if((gamepad.Buttons &
GamePadButtons.Cross) ==
GamePadButtons.Cross)
      done = true;
    if((gamepad.Buttons &
GamePadButtons.Left) ==
GamePadButtons.Left)
      world *=
Matrix4.RotationY(FMath.Radians(1));
```

```
    if((gamepad.Buttons &
GamePadButtons.Right) ==
GamePadButtons.Right)
      world *= Matrix4.RotationY(-
FMath.Radians(1));

    model.SetWorldMatrix(ref world);


graphics.Enable(EnableMode.Blend);

graphics.SetBlendFunc(BlendFuncMode.Add,
BlendFuncFactor.SrcAlpha,
BlendFuncFactor.OneMinusSrcAlpha);

graphics.Enable(EnableMode.CullFace);

graphics.SetCullFace(CullFaceMode.Back,
CullFaceDirection.Ccw);

graphics.Enable(EnableMode.DepthTest);

graphics.SetDepthFunc(DepthFuncMode.LEqual,
true);

    model.Update();
    model.Animate(0.01f);

    sword.Update();

    model.Draw(graphics,program);
    sword.SetWorldMatrix(ref
swordMatrix);
    sword.Draw(graphics,program);
    var swordMatrix =
model.Bones[24].WorldMatrix;
    graphics.SwapBuffers ();
  }
  model.Dispose();
```

```
            program.Dispose();
            graphics.Dispose();
        }
```

Hit *F5* to run and you should see the following screenshot:

Our now familiar walker model will be animated on screen. Now, however, he is holding a sword that will animate with him! Use left and right on the d-pad to rotate the screen and hit **X** to exit the app.

How it works...

We start off loading an additional model, `sword.mdx`. In our game loop, just like with our model, we need to call `Update()` and `Draw()` for our sword. Since the sword model is not animated, we do not need to call the `Animate()` method. We can use the same program and graphics context for both models. The biggest change here is that we find the wrist bone (which happens to be the 24th bone, but we could have also found it by its name) within the model's `Bones` array, and set our sword's `world` matrix to be the same as the wrist bone. This means that as the wrist moves, so will the sword.

There's more...

Each bone contains a fair amount of information. In addition to the `world` matrix and name, each bone also has a bounding sphere, the index in the `Bones` array of its parent, translation and scaling vectors, a rotation quaternion, as well as an array of the parts it modifies.

In addition to an array of `Bones` and `Motions`, `BasicModel` contains a number of other arrays: `Textures`, which contains the textures applied to the model; `Programs`, which contains the `BasicPrograms` attached to the model; `Materials`, which contains the material attributes of the model (such as shininess, emission, opacity, and so on); and `Parts`, which actually contains the vertex information. Oddly, the model's `Motion` data include `Fcurves`, but has no functionality for blending animations.

Chapter 9. Finishing Touches

In this chapter we will cover:

- Opening and loading a web browser
- Socket-based client and server networking
- Accessing (Twitter) data over the network using REST and HttpWebRequest
- Copying and pasting using Clipboard
- Embedding and retrieving a resource from the application assembly
- Configuring your application using PublishingUtility
- Creating downloadable content (DLC) for your application

Introduction

In this chapter we are going to cover the best of the rest. We will be covering a fairly wide range of topics, from networking and copy and paste to preparing your application for deployment.

Opening and loading a web browser

This recipe is by far the book's shortest and simplest recipe. We are going to illustrate how to open the device's web browser and load a web page.

Getting ready

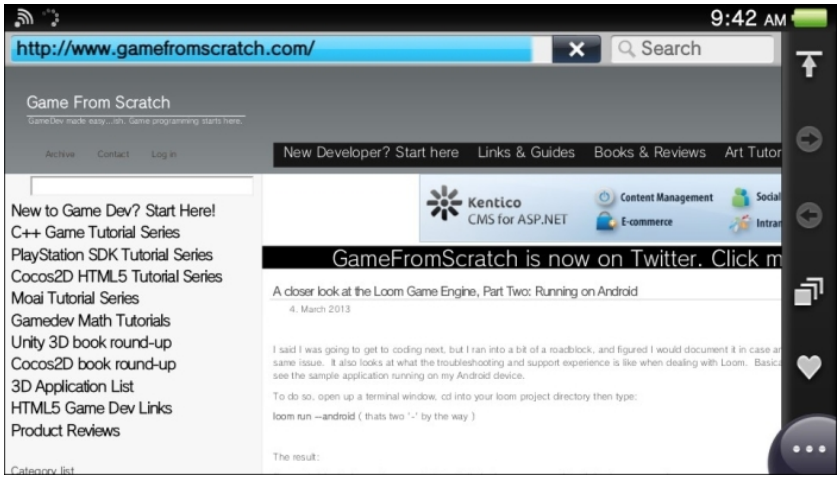
Open **PlayStation Mobile Studio** and create a new project. You can download the full sources as `Ch9_Example1`.

How to do it...

Open `AppMain.cs` and replace the code with the following:

```
using System;
using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Environment;

namespace Ch9_Example1 {
  public class AppMain {
    public static void Main (string[]
args)  {
      var action =
Shell.Action.BrowserAction(@"http://
www.gamefromscratch.com");
      Shell.Execute(ref action);
    }
  }
}
```

Run the application and you should see the following screenshot:



A web browser will open on the user's device and load the world's greatest website!

How it works...

The `Shell` class is located in the `Sce.PlayStation.Core.Environment` namespace. As of right now `BrowserAction` is the only available action. As I said earlier, this is a very short recipe.

There's more...

`BrowserAction` will open the Vita browser if run on PlayStation Vita. In the simulator, it will open the default browser on your PC. On Android it will create a browser intent, prompting you which browser you wish to open.

Socket-based client and server networking

This recipe demonstrates socket-based networking between PlayStation Mobile devices. It is a little bit different from prior recipes in that there are two code examples in a single recipe, the client and the server. The client simply sends a message to the server; the server receives and displays any incoming messages on screen.

Getting ready

Open two copies of **PlayStation Mobile Studio** and in each one create a new project. We are going to make use of the UI classes, so you need to add a reference to `Sce.PlayStation.HighLevel.UI` in each project. They are available as `Ch9_Example2` and `Ch9_Example2b`.

How to do it...

First we will create our simple server. Open `AppMain.cs` in the first project and enter the following code:

```
using System;
using Sce.PlayStation.Core;
using Sce.PlayStation.Core.Graphics;
using Sce.PlayStation.Core.Input;
using Sce.PlayStation.HighLevel.UI;
using System.Net;
using System.Net.Sockets;

namespace Ch9_Example2 {
  public class AppMain {

    private static Socket socket;
    private static Label results;
```

```
    private static void Main(string
[] args) {
      var graphics = new
GraphicsContext();
      UISystem.Initialize(graphics);

      var scene = new Scene();
      results = new Label() { Text =
"Waiting for connection on IP ",
        Width =
graphics.Screen.Width, Height =
graphics.Screen.Height};
      var panel = new Panel() {
Width=graphics.Screen.Width,
Height=graphics.Screen.Height };
      panel.AddChildFirst(results);

scene.RootWidget.AddChildFirst(panel);
      UISystem.SetScene(scene);

      IPHostEntry ipHostInfo =
Dns.GetHostEntry(Dns.GetHostName());
            IPAddress ipAddress =
ipHostInfo.AddressList[0];

      results.Text +=
ipAddress.ToString() + " on PORT
9210";

      IPEndPoint ipEndPoint = new
IPEndPoint(IPAddress.Any,9210);
      socket = new
Socket(AddressFamily.InterNetwork,SocketType.Stre
      socket.Bind(ipEndPoint);
      socket.Listen(10);

      socket.BeginAccept(new
AsyncCallback(IncomingSocket),null);
      bool quit = false;
      while(!quit) {
```

```
if((GamePad.GetData(0).Buttons &
GamePadButtons.Cross) ==
GamePadButtons.Cross)
          quit = true;
        graphics.Clear();

UISystem.Update(Touch.GetData(0));
        UISystem.Render();
        graphics.SwapBuffers();
      }
      if(socket.Connected)
        socket.Disconnect(false);
      socket.Close();
    }

    private static void
IncomingSocket(IAsyncResult
asyncResult)
    {
      using(var incomingSocket =
socket.EndAccept(asyncResult)){
        var data = new byte[1024];
        var bytesReceived =
incomingSocket.Receive(data);
        results.Text += "\n" +
System.Text.Encoding.ASCII.GetString(data,0,bytes

      }
      socket.BeginAccept(new
AsyncCallback(IncomingSocket),null);
    }
  }
}
```

This code comprises our extremely primitive server.

Now in the other project, replace the code in `AppMain.cs` with the following code:

```
using System;
using Sce.PlayStation.Core;
using Sce.PlayStation.Core.Graphics;
using Sce.PlayStation.Core.Input;
using Sce.PlayStation.HighLevel.UI;
using System.Net;
using System.Net.Sockets;

namespace Ch9_Example2b {

  public class AppMain {

    private static Socket socket;
    private static EditableText
textIpAddress;
    private static EditableText
textMessage;

    private static void Main(string
[] args) {
      var graphics = new
GraphicsContext();
      UISystem.Initialize(graphics);
      var scene = new Scene();
      textIpAddress = new
EditableText() { Width=
graphics.Screen.Width, Height = 32,
DefaultText= "Enter IP Address" };
      textMessage = new
EditableText() { Y=40, Width=
graphics.Screen.Width, Height = 32,
DefaultText= "Message to send" };
      var buttonGo = new Button() {
Y=80, Width=300, Height=32, Text="Go"
};

      buttonGo.ButtonAction +=
HandleButtonGoButtonAction;
      var panel = new Panel() {
Width=graphics.Screen.Width,
```

```
Height=graphics.Screen.Height };

      panel.AddChildFirst(buttonGo);

panel.AddChildFirst(textMessage);

panel.AddChildFirst(textIpAddress);


scene.RootWidget.AddChildFirst(panel);

      UISystem.SetScene(scene);

      bool quit = false;
      while(!quit) {

if((GamePad.GetData(0).Buttons &
GamePadButtons.Cross) ==
GamePadButtons.Cross)
          quit = true;
        graphics.Clear();

UISystem.Update(Touch.GetData(0));
        UISystem.Render();
        graphics.SwapBuffers();
      }
    }

    static void
HandleButtonGoButtonAction (object
sender, TouchEventArgs e)
    {
      using(Socket socket = new
Socket(AddressFamily.InterNetwork,SocketType.Stre
{
        socket.Connect(new
IPEndPoint(IPAddress.Parse(textIpAddress.Text),92

socket.Send(System.Text.ASCIIEncoding.ASCII.GetBy
        socket.Close();
```

```
            }
        }
    }
}
```

Now run the first application (your server) and make note of the IP address. Run the other application (leaving the first application running) and enter this IP address, as well as a message, and click on **Go**, like this:

Now if you look back at your server window you should see the following:

As you can see, communicating between devices is relatively simple. The server can actually serve requests from a number of different devices and handle a number of concurrent requests. Click on the **X** button on either application to exit.

How it works...

Let's start off looking at our server code. We start off creating our GUI by creating a `GraphicsContext`, which is used to initialize the `UISystem` singleton, creating a scene and populating it with a panel containing a full screen label for displaying our text. We set the panel as the scene's `RootWidget`, then set the scene running with a call to `SetScene`.

Next we get IP information about the machine by getting the host name with a call to `Dns.GetHostName()` and in turn passing that value to `Dns.GetHostEntry()`, which returns `IPHostEntry`. Because a machine can have multiple IP addresses, especially because of IPv4 and IPv6, `IPHostEntry` has an array of `IPAddress` values, but we are only interested in the first one. Please note that although the vast majority of the time the first address returned will be the IPv4 address, it is not safe to assume so! Now that we have our machine's IP address, we display it on the `Label` text, as well as the port number, 9210.

Next we create an `IPEndPoint`; the values we pass as parameters are the IP addresses and port that we will accept connections on. We pass `IPAddress.Any` and 9210, meaning we will accept connections from any IP address that comes in on port 9210. Next we create our `Socket`. When creating a `Socket` there are a ton of available connection options and protocols—literally thousands of possible combinations. `AddressFamily.InterNetwork` is a very long way of saying we are using IPv4; this is going to be a streaming socket using the TCP protocol. Next we call `Bind()`, which associates the socket with the `IpEndPoint`. You need to perform a `Bind` operation before you can call `Listen` on a port. Speaking of `Listen()`, that's exactly what we do next, call `Listen` on the socket, which will listen for network activity. The value 10 is the maximum length of the queue for incoming socket requests and is a value I chose pretty much at random. Finally, when calling `BeginAccept` we register an `AsyncCallback` function, `IncomingSocket`, which will be called each time a socket connection is established.

Next is our `Main` loop; we check to see if the user hit the **X** button, in which case we exit the program. We also call all the required functions, clearing the screen, updating the UI, and rendering the UI to screen. Finally, after our event loop exits, we disconnect and close our `Socket`. Until a socket is closed, you cannot connect to it again, so be sure to call `Close()` once complete.

Finally, we have the `IncomingSocket` function, which is called each time `BeginAccept` is called. We then call `EndAccept` on the socket, passing in the `IAsyncResult` value that was passed to our function. `EndAccept` returns the actual data that was sent over the socket, which we copy to a `byte[]` buffer by calling `Receive()`. We then convert this buffer to a string and display it by appending it to our `Label` text. We then call `BeginAccept` again, preparing it for the next incoming socket to repeat the entire process all over again.

This handles our basic server code. Now let's take a quick look at what we did in the client code. This code starts off by creating a very simple GUI containing an `EditableText` widget for IP address entry and another for entering the message to send. We also create

a `Button` labeled **Go** and call the function `HandleButtonGoButtonAction` when the button is clicked. The remaining code simply adds the controls to the scene, sets the scene active with a call to `UISystem.SetScene()`, and then creates an identical loop to the one we used for the server.

In the `HandleButtonGoButtonAction` function, we create a `Socket` in the same way we did on the server. A client, however, does not need to call `Bind()` for the socket before using it. We connect to our server using `socket.Connect`, creating an `IPEndPoint`, from an `IPAddress` that was created by parsing the text the user entered. Of course, there is no error handling code here, but in a production environment you would obviously improve this code a great deal. Once connected, sending data is as simple as sending a `byte[]` array in a call to `socket.Send()`. We simply convert the message the user typed in to a `byte[]` array and send it. Finally, we close the socket. If you forgot to close it, the next call to `HandleButtonGoButtonAction` would fail. You will notice that `Socket` is wrapped in a using block; this is because `Socket` is an `IDisposable` object and will leak resources if not disposed of properly.

There's more...

In both of these examples, we are using `System.Net.Socket`, which is part of .NET Framework, so the same code will run on Windows, Mac, and so on, minus of course the UI-specific parts. Networking is a gigantic subject, one worthy of its own book, so we can only show the simplest of examples here. Sockets also aren't your only option; you can also make use of the higher level `UdpClient` and `TcpClient` classes, which abstract away a lot of the low-level details. There are also options such as `UdpListener` and `TcpListener` for implementing servers. Unless you have a good reason not to, you are better served working at a higher level of abstraction when possible.

Note

Why port 9210?

Absolutely no good reason. I simply picked it because as far as I know it is not a port that is commonly used. A port can be any value from 1 to 65,536, although most of the lower numbers are commonly used for other tasks, such as Port 21 for e-mail, or port 80 for HTTP traffic. I used to always use port 4242 for my random port usage, but it turns out programmers really like this number for some odd reason…

See Also

- Refer to Chapter 6, *Working with GUIs*, for more details on working with the UI library. See http://bit.ly/SKWx2b for more details on `System.Net.Socket` usage.

Accessing (Twitter) data over the network using REST and HttpWebRequest

You will often want to access data from the Internet, for a variety of different reasons. REST is an incredibly popular format that web services use to expose information online. This recipe illustrates how to use the `HttpWebRequest` class to access one such web service, Twitter.

Getting ready

Open **PlayStation Mobile Studio** and create a new project. You also need to add references to `Sce.PlayStation.HighLevel.UI`, `System.Xml`, and `System.Xml.Linq`. You can download the full source code as `Ch9_Example3`.

How to do it...

Open `AppMain.cs` and replace the code there with the following code:

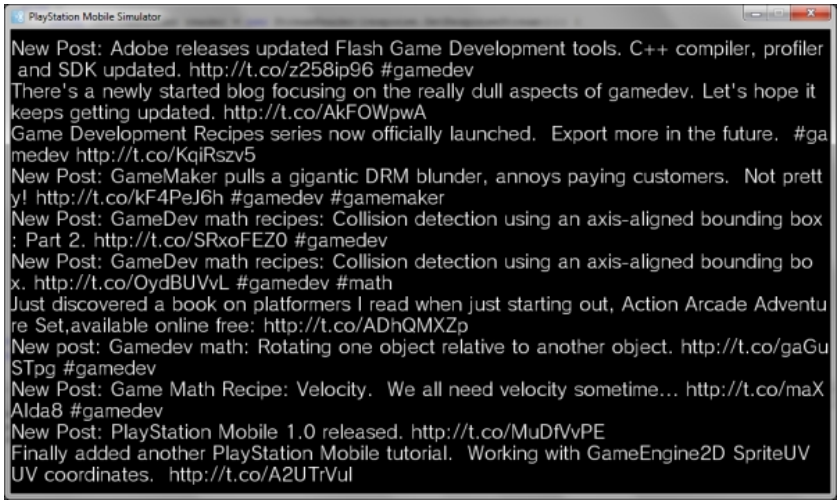
```
using System;
using System.Collections.Generic;
using Sce.PlayStation.Core;
using Sce.PlayStation.Core.Graphics;
using Sce.PlayStation.Core.Input;
using Sce.PlayStation.HighLevel.UI;
using System.IO;
using System.Net;
using System.Linq;
using System.Xml.Linq;

namespace Ch9_Example3 {
```

```
  public class Tweet {
    public string TweetText;
    public string ProfileImage;
    public string Name;
  }

  public class AppMain {
      private static List<Tweet>
tweets = new List<Tweet>();

    private static void Main(string
[] args) {
      var graphics = new
GraphicsContext();
      UISystem.Initialize(graphics);
      var scene = new Scene();
      var results = new Label() {
Text = "",
        Width =
graphics.Screen.Width, Height =
graphics.Screen.Height};

      var panel = new Panel() {
Width=graphics.Screen.Width,
Height=graphics.Screen.Height };

      panel.AddChildFirst(results);

scene.RootWidget.AddChildFirst(panel);
      UISystem.SetScene(scene);

          var request =
HttpWebRequest.Create(@"http://
api.twitter.com/1/statuses/
user_timeline.xml?
id=gamefromscratch");
          request.ContentType =
"application/xml";
          request.Method = "GET";
      request.Timeout = 30000;
```

```
          using (HttpWebResponse
response = request.GetResponse() as
HttpWebResponse) {
              if (response.StatusCode
!= HttpStatusCode.OK)

Console.Out.WriteLine("Error fetching
data. Server returned status code:
{0}", response.StatusCode);
        else {
                using (StreamReader
reader = new
StreamReader(response.GetResponseStream()))
{
                    var content =
reader.ReadToEnd();
                    if
(string.IsNullOrWhiteSpace(content))
{

System.Diagnostics.Debug.WriteLine("Twitter
returned no tweets");
                    }
                    else {
                        XDocument doc
= XDocument.Parse(content);
                        tweets =
(from e in
doc.Root.Descendants("status")

select new Tweet {

    Name= "GameFromScratch",

    ProfileImage =
e.Element("user").Element("profile_image_url").Va

    TweetText =
e.Element("text").Value,
```

```
}).ToList();
                            foreach (var
tweet in tweets) {

results.Text += tweet.TweetText +
"\n";
                            }

                        }
                    }
            }
              }

        bool quit = false;
        while(!quit) {

if((GamePad.GetData(0).Buttons &
GamePadButtons.Cross) ==
GamePadButtons.Cross)
              quit = true;
            graphics.Clear();

UISystem.Update(Touch.GetData(0));
            UISystem.Render();
            graphics.SwapBuffers();
        }
      }
    }
}
```

Run the application and you should see the following screenshot:



Assuming you have a working network connection, you should now see a text dump of a certain person's Twitter feed. Once again, you can exit the application by pressing the **X** button.

How it works...

Twitter returns a great deal of data, but we are only interested in a portion of it, so we create the `Tweet` class to hold just the information we care about. Next we declare a `List` of Tweet objects.

In `Main` we start off with the typical initialization code. In this example we are making use of `UISystem`, so we need to initialize the singleton, and create a scene, a panel, and a full screen label named `results` to hold our text output. We add the label to the panel, add the panel to the scene, and then set the scene active with a call to `UISystem.SetScene()`.

Now we create our `HttpWebRequest` object, passing it the URL http://api.twitter.com/1/statuses/user_timeline.xml?id=gamefromscratch. This is the web service URL you use to request the latest status updates for Twitter user `gamefromscratch` (I wonder who that is?). See the *See also* section of this recipe for a link with more details about the various URLs Twitter supports. We specify that we expect the results to be

XML by specifying the `ContentType` as "application/html", that the request will be a `GET` request (as opposed to a `POST` request) and finally that we want this request to wait for at least 30 seconds before timing out.

Next, we actually make our request by calling `GetResponse()`. As you can see it is wrapped in a using block; this is because `HttpWebResponse` is an `IDisposable` object, so we don't want to leak resources. First, we check the `StatusCode` and print out an error to the console if we get a response other than **OK**. Otherwise, we read the results using `GetResponseStream()` using `StreamReader` (which is also an `IDisposable` object). We read the entire buffer using `ReadToEnd()` and verify that it isn't empty.

At this point we should have a valid XML document in memory, which we read using `XDocument.Parse()` and extract the appropriate values to populate our tweet list. We then loop through the tweet list and print the tweets to our label results. The remaining code is the standard application loop and should be completely familiar at this point.

There's more...

Twitter is just one of many services you can access using `HttpWebRequest` and `HttpWebRequest` is just one of the ways you can use to access information over the Web. The PlayStation Mobile SDK also includes the `ServiceModel` assemblies, which allow you to access SOAP-based web services. SOAP-based web services are becoming increasingly less common as REST services grow in popularity.

Also be aware that the Twitter API we used is incredibly limited. It is only able to access publically available information and is read-only. Additionally, it has a limited rate of 150 requests per hour. You can, however, register your application with Twitter to have the rate limits raised and to be able to access additional information and functionality, such as publishing a tweet. This process requires authenticating, which is far too complex to go into more detail here.

Note

REST?

REST stands for **Representational State Transfer** and is a software design architecture style, most commonly implemented over HTTP, providing what are commonly referred to as RESTful services. In a grossly simplified description, you generally access a REST service by creating a specially formed URL and making an HTTP request. No information about the client session state is stored on the server, so each request must then contain all information the request requires. The lack of server-side session state makes it much easier to scale up such services. This lack of session state, though, is a double-edged sword, making authentication, for example, a much more complex issue than with traditional web services.

See also

- See http://bit.ly/Q6uoWL for more details on the Twitter developer program if you wish to add Twitter functionality to your application
- Refer to Chapter 6, *Working with GUIs*, for more details on using `UISystem` and the PlayStation Mobile UI system

Copying and pasting using Clipboard

PlayStation Mobile provides a simple `Clipboard` class to provide copy and paste functions. This recipe shows how to add basic copy and paste support to your application.

Getting ready

Open **PlayStation Mobile Studio** and create a new project. We are going to make use of the UI system for this recipe, so be sure to add a reference to `Sce.PlayStation.HighLevel.UI`. The full project is available as `Ch9_Example4`.

How to do it...

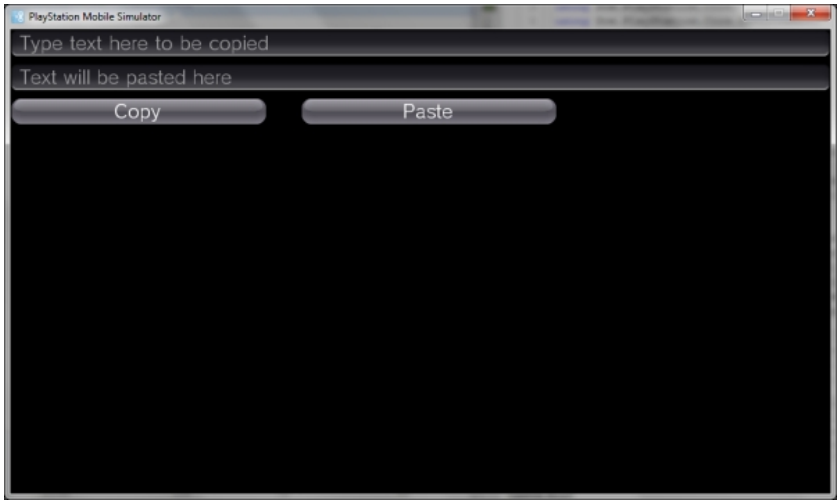
Open `AppMain.cs` and replace the code there with the following code:

```
using System;
using System.Collections.Generic;
using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Environment;
using Sce.PlayStation.Core.Graphics;
using Sce.PlayStation.Core.Input;
using Sce.PlayStation.HighLevel.UI;

namespace Ch9_Example4 {

  public class AppMain {

  private static void Main(string []
args){

      var graphics = new
```

```
GraphicsContext();
      UISystem.Initialize(graphics);
      var scene = new Scene();

      var textCopied = new
EditableText() { Width=
graphics.Screen.Width, Height = 32,
DefaultText= "Type text here to be
copied" };
      var textPasted = new
EditableText() { Y=40, Width=
graphics.Screen.Width, Height = 32,
DefaultText= "Text will be pasted
here" };
      var buttonCopy = new Button() {
Y=80, Width=300, Height=32,
Text="Copy" };
      var buttonPaste = new Button()
{ X=340, Y=80, Width=300, Height=32,
Text="Paste" };
      buttonCopy.ButtonAction +=
(sender, e) => {

Clipboard.SetText(textCopied.Text);
      };
      buttonPaste.ButtonAction +=
(sender, e) => {
        textPasted.Text =
Clipboard.GetText();
      };
      var panel = new Panel() {
Width=graphics.Screen.Width,
Height=graphics.Screen.Height };


panel.AddChildFirst(buttonCopy);

panel.AddChildFirst(buttonPaste);

panel.AddChildFirst(textCopied);
```

```
panel.AddChildFirst(textPasted);


scene.RootWidget.AddChildFirst(panel);

      UISystem.SetScene(scene);

      bool quit = false;
      while(!quit) {

if((GamePad.GetData(0).Buttons &
GamePadButtons.Cross) ==
GamePadButtons.Cross)
          quit = true;
        graphics.Clear();

UISystem.Update(Touch.GetData(0));
        UISystem.Render();
        graphics.SwapBuffers();
      }
    }
  }
}
```

Run this application and you should see the following screenshot:



Two text fields and two buttons appear on screen. Enter text in one field and click on **Copy**, then on **Paste**, and the other field should be populated with the text you just entered. Once again, you can click on **X** to exit.

How it works...

This recipe starts off by initializing the `GraphicsContext` and `UISystem` as usual. We create a UI `Scene`, two `EditableText` fields, one for entering the text to copy, the other to receive the text when pasted. We also create a pair of `Button` objects for the **Copy** and **Paste** buttons. We then register a handler for when the **Copy** button is pressed, inside which we copy the text to the clipboard using the `Clipboard.SetText` method, passing the `Text` value of the `textCopied EditableText` field as the text to be copied. We then register a handler for the **Paste** button's click action, where we assign the text from the clipboard to the `Text` value of `textPasted`. The text is retrieved from the clipboard using the `Clipboard.GetText()` method.

The remaining code in this recipe should be completely familiar at this point; we add all of our UI elements to a `Panel` object and set the panel as the root element of the scene, which we then make active. Otherwise it's a standard event loop that continues until the user clicks on the **X** button.

There's more...

The `Clipboard` object only handles copying and pasting of text data. The `Clipboard` object is declared in the `Sce.PlayStation.Core.Environment` library.

Embedding and retrieving a resource from the application assembly

It is possible to store data directly in the application assembly, instead of as a separate file. This recipe demonstrates how to store and retrieve an image resource from the application assembly.

Getting ready

Open **PlayStation Mobile Studio** and create a new application. You will need to include the `Sce.PlayStation.HighLevel.GameEngine2D` library reference. For the image file, I will be re-using our trusty **FA-18H** sprite, although you of course can use whatever image you wish. You can download this complete recipe as `Ch9_Example5`.

How to do it...

In your newly created solution, add the image file to your project. Right-click on the image file and select **Build Action** | **EmbeddedResource**.

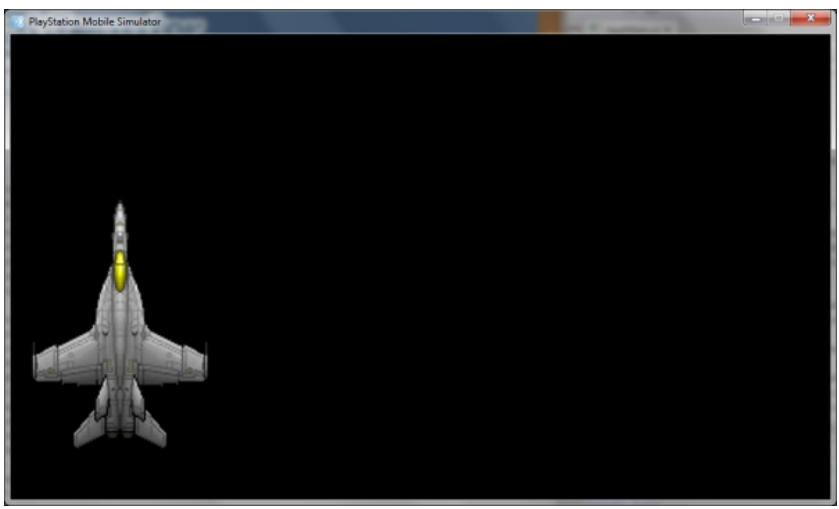
Now open `AppMain.cs` and replace the code there with the following code:

```
using System;
using System.Collections.Generic;

using Sce.PlayStation.Core;
using
Sce.PlayStation.Core.Environment;
using Sce.PlayStation.Core.Graphics;
using Sce.PlayStation.Core.Input;
using
```

```
Sce.PlayStation.HighLevel.GameEngine2D;
using
Sce.PlayStation.HighLevel.GameEngine2D.Base;
using System.Reflection;
using System.IO;

namespace Ch9_Example5 {
  public class AppMain {
    public static void Main(string[]
args)  {
      Director.Initialize();
      Scene scene = new Scene();

scene.Camera.SetViewFromViewport();

      Assembly resourceAssembly =
Assembly.GetExecutingAssembly();
      if
(resourceAssembly.GetManifestResourceInfo("Ch9_Ex
== null) {
          throw new
FileNotFoundException("File not
found.");
      }

      Stream fileStream =
resourceAssembly.GetManifestResourceStream("Ch9_E
      Byte[] dataBuffer = new
Byte[fileStream.Length];

      fileStream.Read(dataBuffer, 0,
dataBuffer.Length);

      Texture2D t = new
Texture2D(dataBuffer,false);
      TextureInfo ti = new
TextureInfo(t);
      SpriteUV sprite = new
SpriteUV(ti);
      sprite.Scale = ti.TextureSizef;
```

```
            fileStream.Close();
            fileStream.Dispose();

            scene.AddChild(sprite);

        Director.Instance.RunWithScene(scene);
            }
        }
    }
```

Run the application and you should see the following:



Your image will be loaded and displayed on screen, positioned in the lower-left corner.

How it works...

To keep the code shorter, we are making use of the `GameEngine2D` library. We start by initializing the `Director` singleton, creating and calibrating a `Scene` object. Next we get a reference to our running application `Assembly` by calling `Assembly.GetExecutingAssembly()`. Next we check to make sure our embedded image exists within the assembly by calling `GetManifestResourceInfo()` passing in the filename prefixed

with the application name `Ch9_Example5.FA-18H.png`. If the file isn't found we throw a `FileNotFoundException`.

Assuming the file is found, we retrieve a `Stream` using `GetManifestResourceStream()`, again passing in the filename prefixed with the application name. Next we create a `byte[]` array big enough to hold the stream contents, then read the stream into our array. We then use this array to create a `Texture2D` object, then use that object to create a `TextureInfo`, which in turn we create a `SpriteUV` with. Now we close and dispose of our `Stream`, add our newly created sprite to the scene, and start things running.

There's more...

For the sake of brevity, this example (and others earlier in this book) don't completely clean up before the application exits, resulting in a very short-lived memory leak. Just be aware that, if you load a `Texture2D` in this manner, you are still responsible for disposing it off.

See also

- See Chapter 3, *Graphics with GameEngine2D*, for more details on working with the `GameEngine2D` library

Configuring your application using PublishingUtility

Once you have finished your application and are getting ready to publish it, there are a number of settings you can configure using PublishingUtility. This recipe takes a closer look at how you configure your application using this tool.

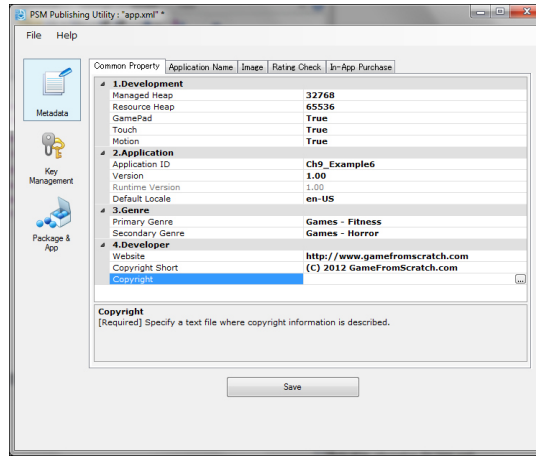
Getting ready

You will need to have an existing project to work through this recipe. Either use a prior recipe's project or create a new one.

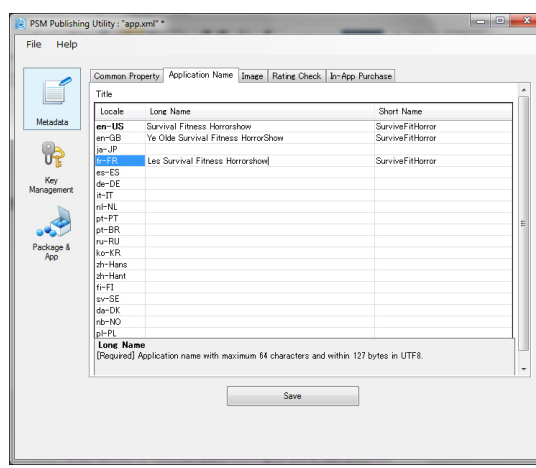
How to do it...

To configure your application using PublishingUtility perform the following steps:

1. Open **PublishingUtility**; a shortcut should be available in the `PlayStation Mobile` folder of your **Start Menu**.
2. In **PublishingUtility**, select **File** | **Load** and browse to the `app.xml` file of the project you are working on, then click on **Open**.
3. With metadata selected on the left-hand side, a series of tabs will appear across the top of the window. Select the **Common Property** tab and enter the relevant information about your application. The primary and secondary genre are used to position your application within the store; the purpose of the settings in the **Development** section are described later on. Fill in a shorthand copyright message as well as provide a TXT file with your complete copyright details (this is mandatory, and will fail if you do not provide a file with a TXT extension). Fill the form out, such as the following example of a horror fitness game:

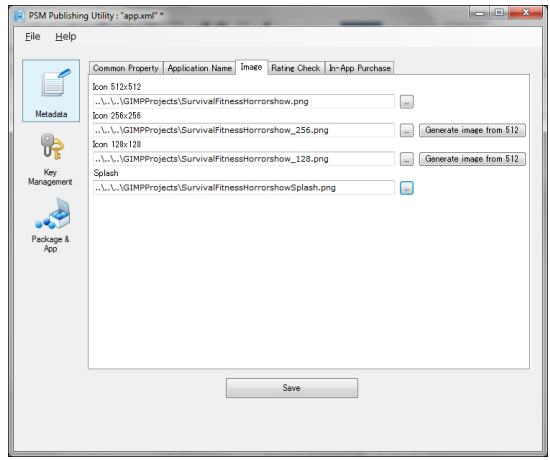


4. Now switch over to the **Application Name** tab and enter a long and short name for your application for each language area you support. Both names can be the same, although the short name is limited to 16 characters while the long name can be as long as 64 characters. The following is our game with values defined for American and British English, as well as French:

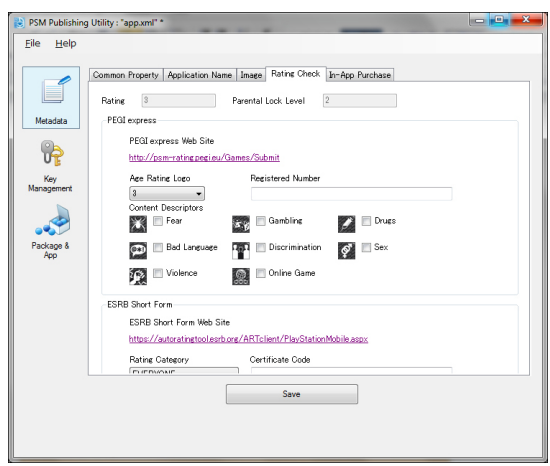


5. Next switch over to the **Image** tab. Here you need to provide at least a 512 x 512 image that will be used to create the applications icons. You can click on the **Generate image from 512** button to have it

automatically create resized versions as well. You can also specify a splash screen, the details of which are provided in the following screenshot:



6. Finally, we come to the **Rating Check** tab. This is where you can apply for various maturity ratings for your game (not applicable for most applications). The requirements for ratings vary from country to country. We will be ignoring the **In-App Purchase** tab for now:



7. Finally, click on **Save** and your `app.xml` file will be updated.

If you open `app.xml`, the code should look something like the following code snippet:

```
<?xml version="1.0" encoding="utf-8"?
>
<application
project_name="Ch9_Example6"
version="1.00" runtime_version="1.00"
sdk_version="1.00.00"
default_locale="en-US">
  <name>
    <localized_item locale="en-US"
value="Survival Fitness Horrorshow" /
>
    <localized_item locale="en-GB"
value="Ye Olde Survival Fitness
HorrorShow" />
    <localized_item locale="fr-FR"
value="Les Survival Fitness
Horrorshow" />
  </name>
  <short_name>
    <localized_item locale="en-US"
value="SurviveFitHorror" />
    <localized_item locale="en-GB"
value="SurviveFitHorror" />
    <localized_item locale="fr-FR"
value="SurviveFitHorror" />
  </short_name>
  <parental_control lock_level="2" />
  <rating_list highest_age_limit="3"
has_online_features="false">
    <online_features chat="false"
personal_info="false"
user_location="false"
exchange_content="false" />
    <rating type="ESRB" value="3"
age="3" code="" />
    <rating type="PEGIEX" value="3"
```

```
age="3" code="" />
  </rating_list>
  <images icon_128x128="..\..\..
\GIMPProjects
\SurvivalFitnessHorrorshow_128.png"
icon_256x256="..\..\..\GIMPProjects
\SurvivalFitnessHorrorshow_256.png"
icon_512x512="..\..\..\GIMPProjects
\SurvivalFitnessHorrorshow.png"
splash_854x480="..\..\..\GIMPProjects
\SurvivalFitnessHorrorshowSplash.png"
/>
  <genre_list>
    <genre value="Games:Fitness" />
    <genre value="Games:Horror" />
  </genre_list>
  <developer>
    <name value="" />
  </developer>
  <website href="http://
www.gamefromscratch.com" />
  <copyright author="(C) 2012
GameFromScratch.com" />
  <runtime_config>
    <memory managed_heap_size="32768"
resource_heap_size="65536" />
  </runtime_config>
  <feature_list>
    <feature value="GamePad" />
    <feature value="Touch" />
    <feature value="Motion" />
  </feature_list>
</application>
```

There's more…

The end result of this entire operation is the generation or modification of your project's `app.xml` file. Many of the actions performed using PublishingUtility can be accomplished by hand editing the XML file. You can make changes in the XML file and they

will persist if you reopen the file in the utility.

The `GamePad`, `Touch`, and `Motion` values can be used to toggle support for the gamepad, touch screen, and motion controls, respectively, on and off. If you turn `GamePad` on for a device that does not support a game pad, the onscreen controller will be displayed.

The `Managed Heap` and `Resource Heap` values control your application's memory allocation. The value is specified in kilobytes, with a maximum total value of 96 MB combined. The managed heap is the data your application will use to run; for example, if you allocate a `byte[]` array, this would consume space on the managed heap. `Resource Heap` on the other hand is the storage used by PlayStation Mobile when loading assets such as audio files and images.

When specifying your images, there are a few things to be aware of. You need to create only a single 512 x 512 image that can be used to generate the smaller icon sizes. The splash screen image needs to be 854 x 480 in size and 8 bits per pixel indexed in format. As of this writing, the tool will currently fail if there is a space in your file path for any of the images.

Note

Using a PNG image for creating a splash screen of 8 bits per pixel can prove to be a bit tricky. In the GIMP photo editor, create or load your image as usual, then select **Image** | **Mode** | **Index…**. Set 255 as the maximum number of colors. The one particular snag you can run into is that if you use less than 16 colors in your image, it will be saved as 4bpp or 2bpp. To prevent this, I added a new layer behind the image, so it isn't visible, and filled it with a multicolor gradient. This gets the image's color count over 16, resulting in an 8-bit image when saved.

See also

- See the *Configuring an Android application to use*

onscreen controls recipe in Chapter 2, *Controlling Your PlayStation Mobile Device*, for more details on enabling the `GamePad` option

- See Appendix, *Publishing Your Application*, for details on publishing your application or generating keys using PublishingUtility

Creating downloadable content (DLC) for your application

The PlayStation Store supports adding downloadable content for your application. This recipe examines how you use PublishingUtility to create DLC items for use with your app.

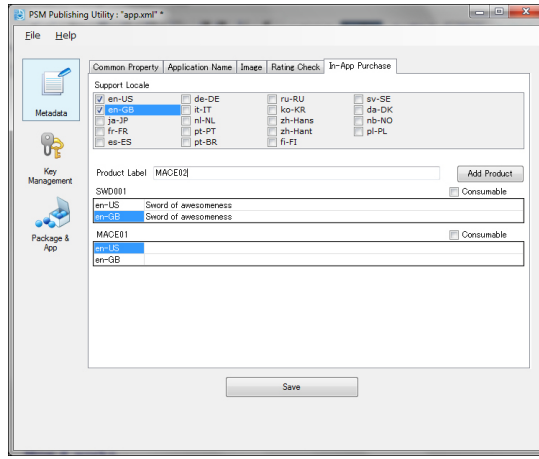
Getting ready

Like the prior recipe, this one assumes you have an existing PSM project to work with.

How to do it...

To create DLC for your application perform the following steps:

1. Open **PublishingUtility**; a shortcut should be available in the `PlayStation Mobile` folder of your **Start Menu**.
2. Open the `app.xml` file of the application, you want to add downloadable content to.
3. Once open, select the **In-App Purchase** tab. To add a new item, enter a six-digit alpha-numeric code in the **Product Label** field. Check each language your DLC is going to be available in and click on the **Add Product** button.
4. An entry will appear for each item you have added, with one line displayed per language specified. In the right-hand side field enter the localized description of the item. If the item is consumable (used up on use) check the **Consumable** checkbox. It should look somewhat like the following screenshot:



5. Click on the **Save** button when done.

How it works...

Just as in the prior recipe, the end result of this process is the generation of the `app.xml` file. Here is the DLC specific output from the preceding example:

```
<purchase>
    <product_list>
      <product label="SWD001"
type="normal">
        <name>
          <localized_item locale="en-
US" value="Sword of awesomeness" />
          <localized_item locale="en-
GB" value="Sword of awesomeness" />
        </name>
      </product>
    </product_list>
  </purchase>
```

The product is actually created on the server when you publish your application. You can control DLC details using the PlayStation Mobile portal.

There's more...

There is some programming involved in actually showing the DLC items to the user for purchase. Essentially, the process involves creating and showing an `InAppPurchaseDialog` to the user. You get a list of purchasable items from the server by calling `GetProductInfo()`; those items are exactly what we created in this process. Next you use `GetTicketInfo()` to get ticket information from the server, which will hold the transaction details. The `Purchase()` method then attempts to make the actual purchase, while `Consume()` actually consumes the item if it's a usage-based consumable item.

See also

- The process of implementing DLC in code happens across multiple parts and doesn't lend itself to the recipe format as a result. Fortunately, there are a pair of samples you can work from: `InAppPurchaseSample1` and `InAppPurchaseSample2`. Assuming you used a default install, they will be located at `C:\Users\Public\Documents\PSM\sample\Services`.
- See Appendix, *Publishing Your Application*, for more details on publishing to the PlayStation Mobile store and accessing the portal.

Appendix A. Publishing Your Application

Introduction

At this point in the book you should know everything you need to create the next big game. It's time to look at the process of publishing your application to the PlayStation store.

Process overview

The following graphic created by Sony details the application development process, from initial creation to getting paid for your hard work. The portions in blue are the parts that you, the developer, are responsible for:

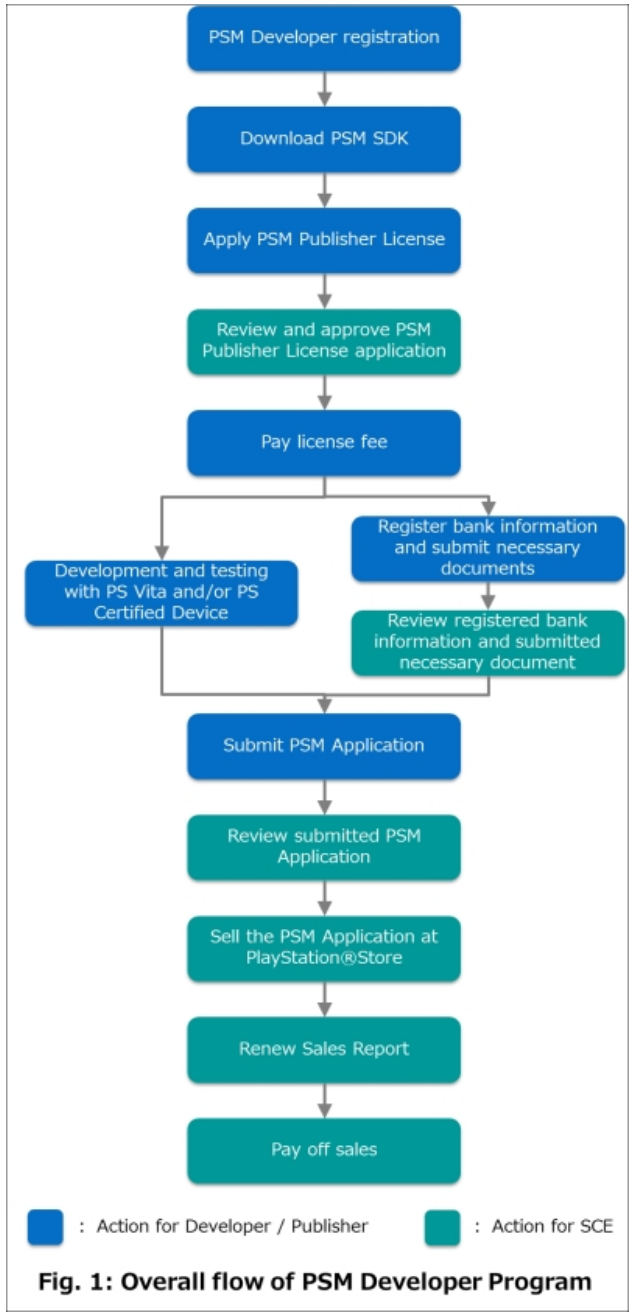


Fig. 1: Overall flow of PSM Developer Program

We have covered the development and testing portions in detail already, earlier in the book. Now we will look at the other details of

publishing your application to the store.

Obtaining a developer license

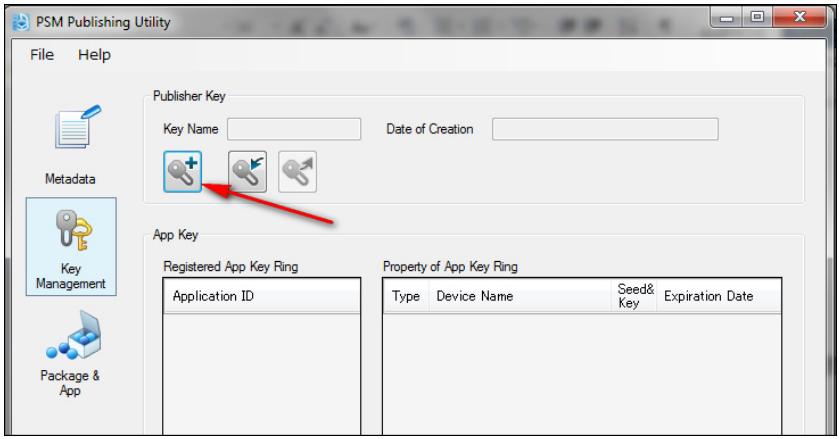
Once you have developed your application and are ready to publish it, you will need to obtain a developer license. As of the time of writing, this costs US $99 a year and is available at https://psm.playstation.net.

If you don't already have one, you need to register for a Sony Entertainment Network account (if you have a PlayStation online or have used any of Sony's forums, this is your SEN account). They are free to sign up for and are required if you want to participate in the PlayStation Mobile development forums.

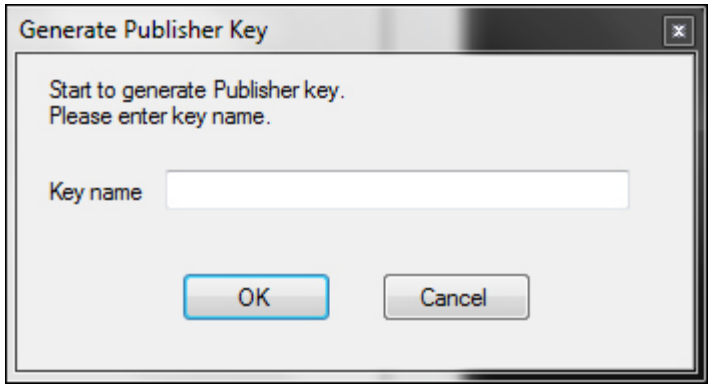
Once you've logged in to the portal, fill in the application form for a developer license, which will be submitted to Sony. Once your application is approved, you will receive an e-mail indicating your acceptance as well as a link that will direct you to the form where you can pay the fee. PlayStation Mobile is now active on your SEN account.

Your publisher keys

Now that your account is activated and approved you can request your publisher keys. Keys are created using PublishingUtility, described earlier in Chapter 9, *Finishing Touches*. Load PublishingUtility and select Key Management. Then, click on the **Generate Publisher Key** button:



A dialog window will appear prompting you for the key name. Enter a value and click on **OK**:



Next, you will be prompted for your SEN ID and password. Enter these values and click on **OK**. You should soon see a dialog stating the keys were successfully created.

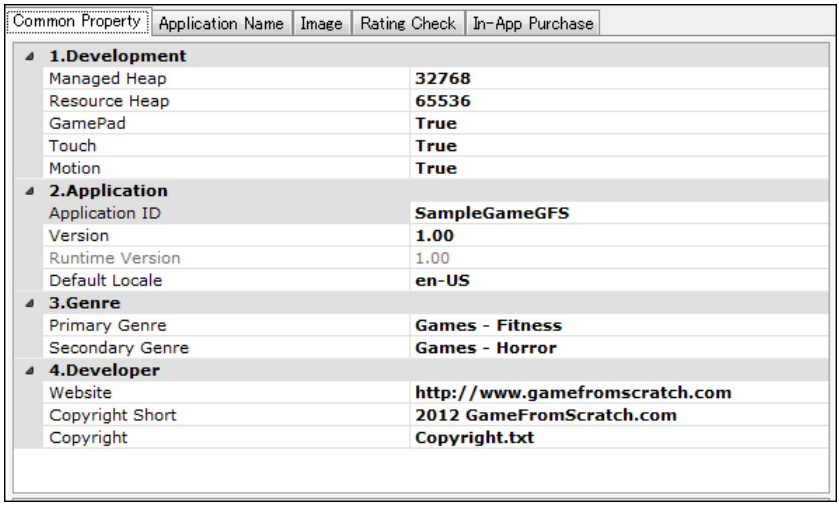
Preparing your application for publication

Before you can deploy your application, you need to configure a number of settings using the PSM Publishing Utility. Open the utility and in the menu select **File** | **Load**, then select your `app.xml` file.

Now select the **Metadata** tab; pretty much every category needs to

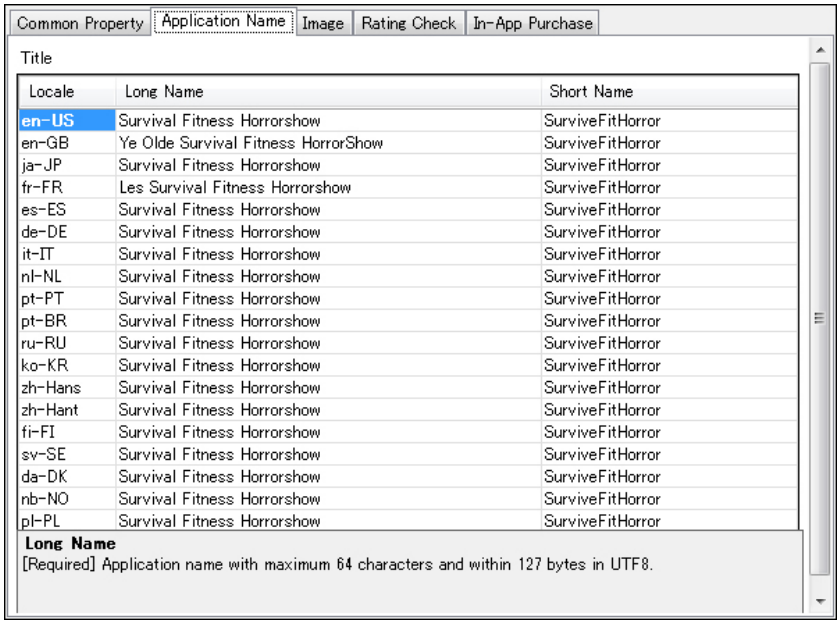
be filled in.

The following screenshot shows a sample entry for the common properties:



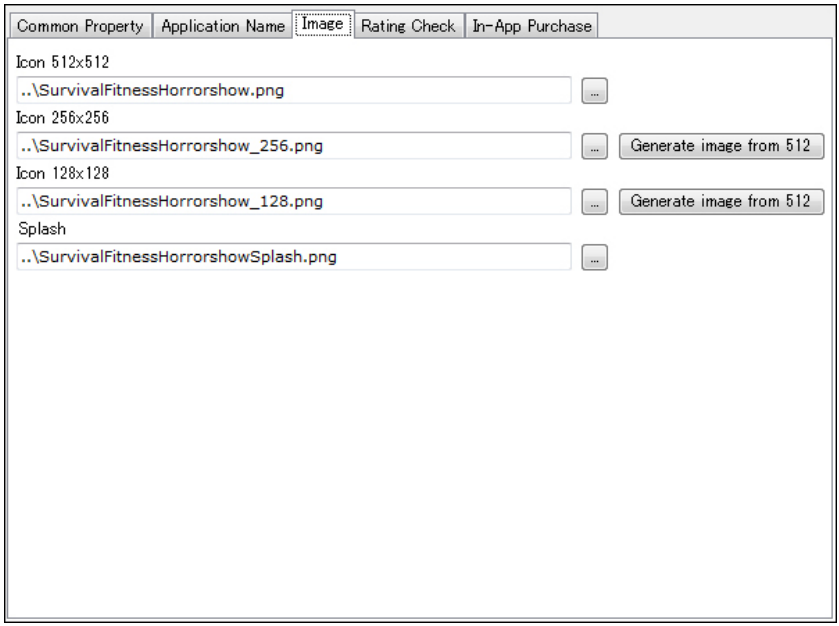
Most of the values here we have already covered or are fairly self-explanatory. Clicking on an item will bring up a description of the values you should use.

Now we look at the **Application Name** tab:



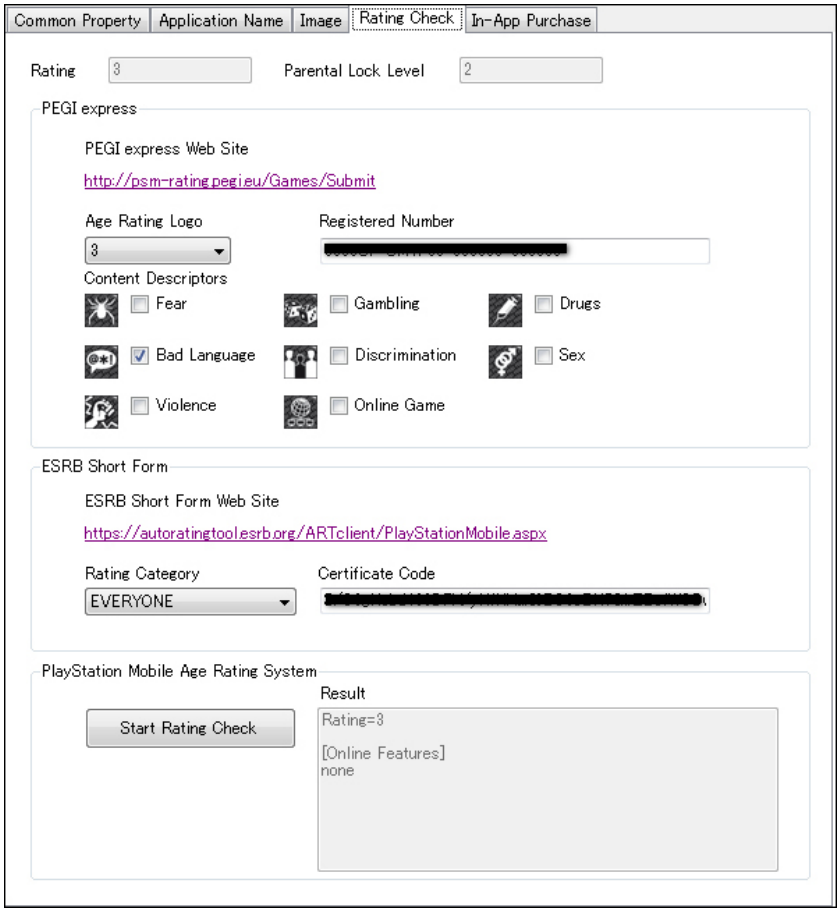
These values simply represent the complete name and short name of your application on the PS Mobile Store. You simply specify the language code as well as the two values you wish to use.

Next up is the **Image** tab:



This part can be a bit tricky. You need to provide a 32 bit per pixel 512x512 resolution image that will be used for your store icon. You can have the tool automatically generate the smaller versions for you. You also need a splash screen that is 854x480 in size and 8 bits per pixel indexed. This last part can be a bit tricky to accomplish. I generated the image using GIMP. However, I had to create an invisible layer with more than 16 colors on it, or the GIMP would automatically reduce the image to 4 bits when saved. It is very important that this image be in the proper format or it will not work.

Finally, let's see the **Rating Check** tab:



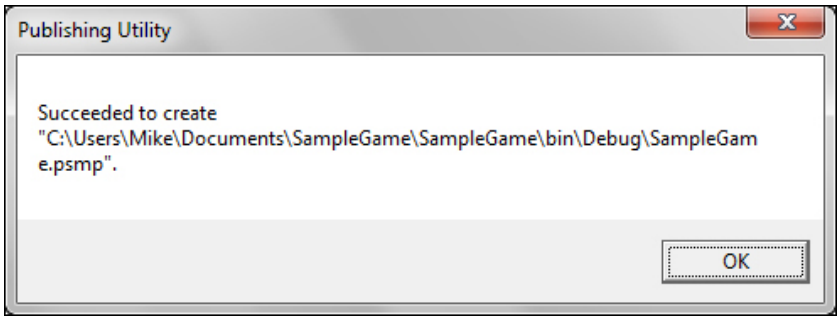
On this page you submit your application to PEGI, ESRB, and the PSM age system. Each service is freely available; follow each link and fill in the resulting forms. In the end you will either be given or e-mailed a code. Once you have the codes, enter them into the corresponding form field.

Now that your application is configured, click on **Save**.

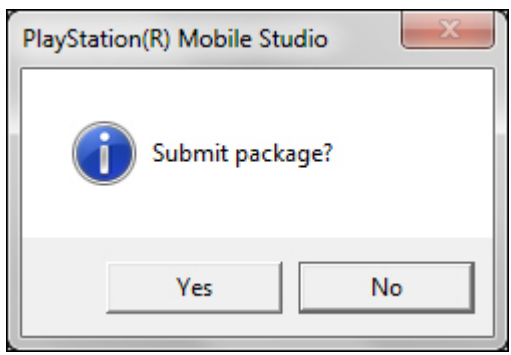
Publishing your application

Open your application in PSM Studio. Do a clean build. Then select the menu **Project**|**Compose PSM Master Package**. If prompted for credentials, log in. Assuming everything goes well, you should see

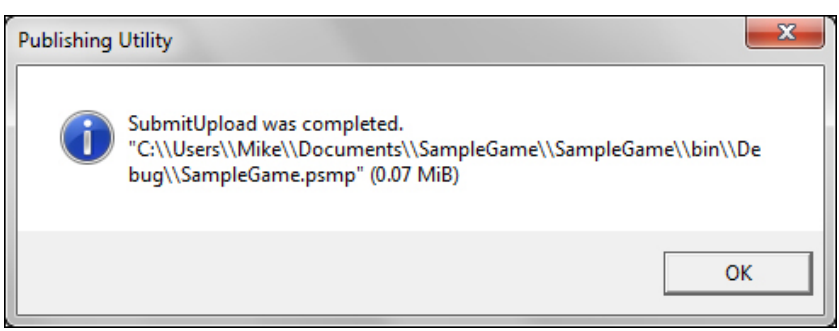
the message shown in the following screenshot:



Next, you will be asked if you wish to submit the application to Sony:



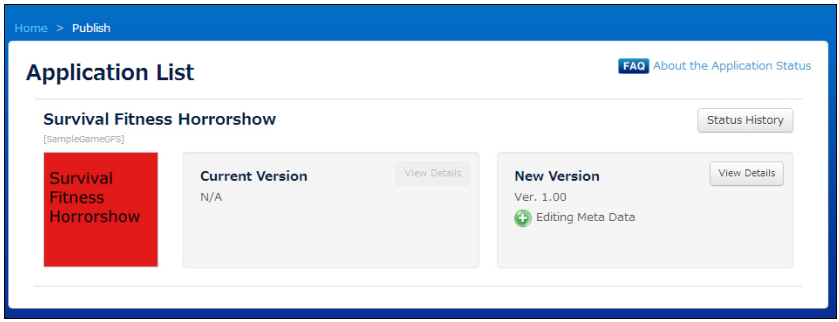
Then, once again assuming everything went OK, you should see the message shown in the following screenshot:



Congratulations, your application has just been submitted to Sony and awaits further approval.

You can now log in to the PSM portal and check on the status of

your application, or make further changes:



You can withdraw your application if required. Once approved, this is where you set pricing information for your application and any DLC you might have included. The portal also has reports for tracking your applications performance in the online store.

Index

A

- Aabb / How it works...
- ActionManager
    - about / Introduction
    - working with / Working with the ActionManager object, How to do it..., How it works...
- Add Product button / How to do it...
- AdHocDraw function / Adding a grid
- AdHocDraw method / How it works...
- analog joysticks
    - about / Using the analog joysticks
    - using / Getting ready, How to do it...
    - working / How it works...
- and (&) operation / How it works...
- Animate() method / How it works...
- animated model
    - sword adding, bones used / Using bones to add a sword to our animated model, How to do it..., There's more...
- application
    - deploying, to Android device / Deploying to PlayStation Mobile certified Android devices, Getting ready, How to do it..., There's more...
    - DLC, creating / Creating downloadable content (DLC) for your application, How it works...
    - publishing / Publishing your application

- application assembly
    - resource, retrieving / Embedding and retrieving a resource from the application assembly, How to do it..., How it works...
    - resource, embedding / Embedding and retrieving a resource from the application assembly, How to do it..., How it works...
- application configuration
    - PublishingUtility, using / Configuring your application using PublishingUtility, How to do it..., There's more…
- Application Name tab / How to do it..., Preparing your application for publication
- application publish
    - process overview / Process overview
    - developer license, obtaining / Obtaining a developer license
    - publisher keys / Your publisher keys
    - preparing steps / Preparing your application for publication
    - steps / Publishing your application
- Apply to layout button / How to do it...

B

- background music
    - playing / Getting ready, How to do it..., How it works..., There's more...
- BasicProgram
    - using, for shader effects / Using BasicProgram to perform texture and shader effects, How to do it..., How it works...,

There's more...
    - using, for texture / Using BasicProgram to perform texture and shader effects, How to do it..., How it works..., There's more...
    - scene lighting control / Controlling lighting using BasicProgram, How to do it...
    - working / How it works..., See also

- BEPU physics engine / There's more...
- bones
    - used, for sword adding / Using bones to add a sword to our animated model, How to do it..., There's more...

- BunjeeEffect / There's more...

C

- camera system
    - implementing / Implementing a simple camera system, Getting ready, How to do it...
    - working / How it works..., See also

- Circle (D key) / How to do it...
- Clipboard.GetText() method / How it works...
- CodePlex / There's more...
- collision
    - detecting / Simple collision detection, How to do it..., How it works...
    - checking / Querying if a collision occurred, How to do it..., How it works...

- common widgets / Common widgets
- compiler / Compiler and runtime
- container widgets / Container widgets
- Control key / How it works...
- copying

    - Clipboard used / Copying and pasting using Clipboard, How to do it..., How it works...

- Crop() function / How it works...
- CSV / There's more...
- currentLanguageId property / How to do it...
- CurrentMotion property / How it works...

D

- 2D particle system
    - creating / Creating a 2D particle system
    - working / How it works...
- 3D model
    - importing, for PlayStation Mobile / Importing a 3D model for use in PlayStation Mobile, How to do it..., How it works..., There's more...
    - displaying in / Loading and displaying a 3D model, How to do it...
    - loading / Loading and displaying a 3D model, How to do it..., How it works..., There's more...
    - displaing in / How it works..., There's more...
    - bones / There's more...
- 3D scene
    - creating / Getting ready, How to do it...
    - working / How it works..., There's more...
- Director function / How it works...
- Dispose() method / How it works...
- DLC
    - creating, for application / Creating downloadable content (DLC) for your application, How it works...

- Draw() method / How it works..., There's more...
- DrawText() function / How it works...
- Dynamic / How it works...

E

- E key / How to do it...
- emulated button / There's more...
- event loop / How it works...
- ExampleScene class / How it works...
- external library
    - using / Building and using an external library, How to do it..., How it works...
    - building / Building and using an external library, How to do it..., How it works...

F

- F5 key / How to do it..., How to do it...
- F18 sprite / How to do it...
- FarSeer physics engine / There's more...
- file storage locations
    - /Application / There's more...
    - /Documents / There's more...
    - /Temp / There's more...
- filesystem
    - working with / Working with the filesystem, How it works..., There's more...
- FMath.Radians()utility function / How it works...
- force
    - applying, to rigid body / Applying force and picking a physics scene object, Getting ready, How to do it..., How it works...

- FPS / How it works...
- fragment shader
    - creating / A fragment (pixel) shader in action, How to do it..., There's more..., See also
    - input semantics / There's more...
    - output semantics / There's more...
- FxComposer / See also

G

- GameEngine2D / Introduction
    - about / Introduction
    - used, for game loop creating / A game loop, GameEngine2D style, Getting ready, How it works..., There's more..., See also
- GameEngine2D application
    - UI library, using / Using the UI library within a GameEngine2D application, How to do it..., How it works...
- game loop
    - creating, GameEngine2D used / A game loop, GameEngine2D style, Getting ready, How it works..., There's more..., See also
- gamepads buttons
    - handling / Handling the controller's d-pad and buttons, How it works..., There's more..., See also
- gamepads d-pad
    - handling / Handling the controller's d-pad and buttons, How it works..., There's more..., See

also

- Generate App Key Ring button / How to do it...
- Generate Publisher Key button / Your publisher keys

H

- HandleButtonGoButtonAction function / How it works...
- Hello World application
    - creating, from HighLevel.UI library / "Hello World" – HighLevel.UI style, How to do it..., There's more...
    - working / How it works...
- HighLevel.UI toolkit / Introduction
- HTC Hero One X
    - system requirements / HTC Hero One X specifications
- HttpWebRequest
    - used, for data accessing / Accessing (Twitter) data over the network using REST and HttpWebRequest, How to do it..., How it works..., See also

I

- image
    - manipulating dynamically / Manipulating an image dynamically, Getting ready, There's more...
- Image tab / Preparing your application for publication
- In-App Purchase tab / How to do it..., How to do it...
- IncomingSocket function / How it works...
- Initialize() function / How to do it...
- Initialize() method / How to do it...

- Input2 wrapper class
  - about / Using the Input2 wrapper class
  - using / Getting ready, How to do it...
  - working / How it works...

J

- joint
  - simulating / Creating a (physics!) joint, Getting ready, How to do it..., How it works...

K

- keys / There's more...
- Kinematic / How it works...

L

- language localization
  - handling / Handling language localization, How to do it..., There's more...

M

- Main() function / How to do it...
- MessageBox dialog
  - about / Displaying a MessageBox dialog
  - displaying / How to do it...
  - working / How it works...
- Metadata tab / Preparing your application for publication
- model
  - animating / Getting ready, How to do it..., How it works...

- ModelConverter
    - file formats / There's more...
- ModelViewer / There's more...
- modes / There's more...
- MonoGame library / There's more...
- motion sensors
    - using / Getting ready, How to do it...
    - working / How it works...
- MoveEffect / There's more...
- multiple animations
    - handling / Getting ready, How to do it..., How it works..., There's more...

N

- .NET libraries / Compiler and runtime

O

- offscreen frame buffer
    - using / Using an offscreen frame buffer to take a screenshot, How to do it..., See Also
- OnEnter() method / How it works...
- OnRestored event / How it works...
- onscreen controls
    - creating, for non-gamepad devices / Creating onscreen controls for devices without gamepads, Getting ready, How to do it…, There's more...
- onscreen controls, on Android device
    - configuring / Configuring an Android

application to use onscreen controls, Getting ready, How to do it…

P

- pasting
  - Clipboard used / Copying and pasting using Clipboard, How to do it..., How it works...
- Physics2D
  - about / Introduction
- physics body toggling
  - between dynamic and kinematic / Switching between dynamic and kinematic, How to do it..., How it works..., There's more...
- physics scene object
  - picking / Applying force and picking a physics scene object, Getting ready, How to do it..., How it works...
- Play() method / How it works...
- PlayStation Mobile
  - about / Introduction
  - portal, accessing / Accessing the PlayStation Mobile portal, How to do it..., How it works...
  - tools, installing / Installing the PlayStation Mobile SDK, How to do it...
  - 3D model, importing / Importing a 3D model for use in PlayStation Mobile, How to do it..., How it works..., There's more...
- PlayStation Mobile Android devices
  - resolutions / There's more...

- PlayStation Mobile certified devices
    - about / PlayStation Mobile certified devices
    - PlayStation Vita, system requirements / PlayStation Vita specifications
    - HTC Hero One X, system requirements / HTC Hero One X specifications
- PlayStation Mobile portal
    - accessing / Accessing the PlayStation Mobile portal, How to do it..., How it works...
- PlayStation Mobile SDK
    - about / A tour of the PlayStation Mobile SDK
    - PSM Studio IDE / PSM Studio IDE
    - runtime / Compiler and runtime
    - compiler / Compiler and runtime
    - UIComposer / UIComposer
    - other utilities / Other utilities
    - certified devices / PlayStation Mobile certified devices
- PlayStation Mobile tools
    - installing / Installing the PlayStation Mobile SDK, How to do it...
- PlayStation Vita
    - system requirements / PlayStation Vita specifications
    - configuring / Deploying to a PlayStation Vita, How to do it..., There's more...
- port 9210 / There's more...
- predefined actions
    - using / Using predefined actions, How to do it...
    - working / How it works...

    - GameEngine2D library actions / There's more...

- PS button / There's more...
- PSM Studio IDE / PSM Studio IDE
- PublishingUtility
    - used, for application configuration / Configuring your application using PublishingUtility, How to do it..., There's more…
- Purchase() method / There's more...

R

- rasterized / There's more...
- Rating Check tab / How to do it..., Preparing your application for publication
- Render() method / How to do it...
- Resize() function / How it works...
- REST
    - used, for data accessing / Accessing (Twitter) data over the network using REST and HttpWebRequest, How to do it..., How it works..., See also
    - about / There's more...
- restitution of coefficient / There's more...
- rigid body collision shapes
    - about / Rigid body collision shapes, How to do it..., How it works...
- RootWidget / Common widgets

S

- Sce.PlayStation.Core.Graphics library / Introduction

- scenes
    - creating / Creating scenes, How to do it..., How it works...
    - grid, adding / Adding a grid
    - sprite, adding / Adding a sprite to a scene, How it works..., There's more...
    - transitioning between / Transitioning between scenes, How to do it..., How it works..., There's more...
    - light, adding / Adding lighting to your scene, How to do it..., How it works..., There's more...
- Scheduler
    - about / Introduction
    - used, for update handling / Handling updates with Scheduler, Getting ready, How to do it...
    - working / How it works...
- server networking / Socket-based client and server networking, How to do it..., How it works...
- SetPixel() function / How it works...
- SetVertices() method / How it works...
- SFX Asset Library / There's more...
- simple game loop
    - creating / Creating a simple game loop, How to do it...
    - working / How it works..., There's more...
- simple simulation, with gravity
    - creating / Getting ready, How to do it..., How it works..., There's more...
- Single Screen tab / How to do it…
- S key / How to do it...
- socket-based client / Socket-based client and server networking, How to do it..., How it works...

- sound effects
    - playing / Getting ready, How to do it..., How it works..., See also
- SoundPlayer class / There's more...
- sprite
    - adding, to scenes / Adding a sprite to a scene, How it works..., There's more...
- SpriteLists
    - using, benefits / Batching a sprite with SpriteLists, How to do it..., How it works..., There's more...
- sprite sheet
    - creating / How to do it...
    - working / How it works...
    - using, in code / Using a sprite sheet in code, Getting ready, How to do it..., How it works..., There's more...
    - batching, with SpriteLists / Batching a sprite with SpriteLists, How to do it..., How it works..., There's more...
- SpritesheetScene / How it works...
- SpriteTile class / There's more...
- Square button (A key) / How to do it...
- Start button / There's more...
- system events
    - handling / Handling system events
    - working / How it works...

T

- Take() method / How it works...
- Text property / Getting ready

- texture
    - creating / "Hello World" drawing text on an image, How to do it..., How it works..., There's more...
    - displaying / "Hello World" drawing text on an image, How to do it..., How it works..., There's more...
- texture 3D object
    - about / Displaying a textured 3D object
    - displaying / How to do it..., How it works..., There's more...
    - limits / There's more...
- textured image
    - translating / Loading, displaying, and translating a textured image, How to do it..., How it works..., There's more...
    - displaying / Loading, displaying, and translating a textured image, How to do it..., How it works..., There's more...
    - loading / Loading, displaying, and translating a textured image, How to do it..., How it works..., There's more...
- textures pixels
    - manipulating / Manipulating a texture's pixels, How it works...
- ToBuffer() function / How it works...
- touch events
    - handling / Handling touch events, How it works...
    - working / How it works..., There's more...
- touch gestures

    - handling / How to do it..., How it works..., See also

- TriangleStrip / There's more...

U

- UI
    - creating, UIComposer used / Creating a UI visually using UIComposer, How to do it..., There's more...
- UIComposer / UIComposer
    - used, for UI creating / Creating a UI visually using UIComposer, How to do it..., There's more...
- UI effect objects / There's more...
- UI library
    - using, within GameEngine2D application / Using the UI library within a GameEngine2D application, How to do it..., How it works...
- UI special effects
    - applying / How to do it..., How it works..., See also
- Update() method / How it works...
- update handling
    - Scheduler, using / Handling updates with Scheduler, Getting ready, How to do it...

V

- vertex shader

    - creating / A vertex shader in action, How to do it...
    - working / How it works..., There's more...
    - input semantics / There's more...
    - output semantics / There's more...

W

- web browser
    - loading / Opening and loading a web browser, How to do it..., How it works...
    - opening / Opening and loading a web browser, How to do it..., How it works...
- widget hierarchies
    - creating / How to do it..., How it works...
    - using / How to do it..., How it works...
    - container widget / Container widgets
    - common widget / Common widgets

X

- X button / How it works..., How to do it...
- X key / There's more...

Z

- Z key / There's more...